Emulating 6502

Harry Fitchett

25717330

25717330@students.lincoln.ac.uk

School of Computer Science

College of Science

University of Lincoln.

Submitted in partial fulfilment of the requirements for the Degree of BSc(Hons) Computer Science

Supervisor: Dr. Charles Fox

April 2024

1 Acknowledgements

Firstly, I want to thank my supervisor Prof. Charles Fox and my fellow students Anthony Goddard and Kristraps Jurkans

2 Abstract

This dissertation explores the process of designing and implementing a Field-Programmable Gate Array (FPGA)-based emulator for the 6502 Instruction Set. The research begins with a comprehensive review of the 6502 architecture, delving into its instruction set, addressing modes, and key features, explores the design considerations and architectural choices made in the development of FPGA emulators. It investigates optimization techniques to enhance performance and resource utilization and validation methods while maintaining compatibility with the 6502 ISA.

3 Table of Contents

4 List of Figures

5 List of Tables

6 Introduction

In 1975, MOS Technology created the MOS 6502 microprocessor. It was designed to be efficient, fast, and most importantly more affordable than its competitors. This led to its use in many well-known devices throughout the 1980's, for example the BBC micro, Commodore 64, and Nintendo Entertainment System. Even today, 6502s are still used as microcontrollers in embedded systems because of their cost effective and efficient design. A more detailed explanation of the 6502, Its design and implementation can be is found from Cox (Cox, 2011)

This project, using Logisim-Evolution, aims to explore various hardware development techniques by creating an emulated 6502 FPGA. This project should serve as an introductory guide to hardware engineering, CPU design and 6502 programming. By the end of this project readers will understand the methods required to emulate the 56 instructions of the MOS 6502 instruction set, the 6 accessible register and their associated 8 flags as described in the 6502 User's Manual (Carr, 1984). Furthermore, Logisim-Evolution will enable users to convert this code into a different hardware description language, VDHL. This could allow users to convert this design to an integrated circuit (IC) layout, which means this project could be compiled into a physical system-on-chip (SOC).

For a detailed introduction to Logisim-Evolution explained see (Self, 2019), an introduction of 6502 programming see (Usborne Computer Books, 1983), and for a technical breakdown of the 6502 see (andkorzh, 2022).

7 Literature Review.

The MOS 6502 processor stands as a pioneering microprocessor in the history of computing. Despite it consisting of only “3,510 transistors and being drawn entirely by hand” (Bagnall, 2006) the 6502 played a pivotal role in the development of home computers and gaming consoles during the late 1970s and early 1980s.

The 6502 was created by MOS Technology, a semiconductor design and fabrication company founded in 1969. Chuck Peddle, the chief designer of the 6502, envisioned a “low-cost microprocessor that could compete with more expensive alternatives like the Intel 8080” Bagnall, B. (2006). In 1975, the MOS 6502 was introduced to the world, priced at just $25, a fraction of the cost of other microprocessors at the time. This affordability made it an attractive option for various applications, especially in the emerging market of personal computers.

One of the earliest adopters of the 6502 was the MOS Kim-1, a “flexible single-board computer” (MOS Technology 1976) released in 1976. While not a commercial success, the Kim-1 showcased the potential of the 6502 and attracted attention from hobbyists and small companies. Later the 6502 was employed in more memorable devices such as the Apple I and II, Atari 2600 and the BBC micro with later variants being used in several devices including the Commodore 64 one of the most successfully selling computer of its time cementing the 6502’s place in computing history.

The 6502’s later variants are still manufactured today mostly for embedded processors appearing in some pacemakers, automobile dashboard and computer keyboards.

7.1 Aims and Objectives.

This project aims to explore the hardware development and CPU design process by

Specific: Designing a circuit using Logisim evolution capable of accurately emulating the behavior of an MOS Technology 6502 Microprocessor, Tailoring components essential to any CPU, such as a clock, to more efficiently emulate the 6502 instruction set.

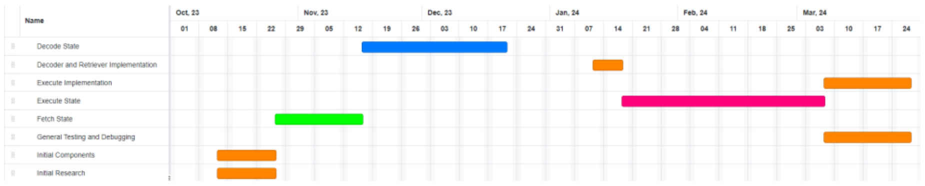
Measurable: Successfully executing example 6502 code, comparing the outcome to the expected example, and finding no discernable differences, Comparing the average execution time of instructions to their execution time when processed by and original 6502 with a decrease in time efficiency no greater than 20%.

Achievable: Utilizing Logisim evolutions accessible and visual development environment to debug components more easily and in real time and leveraging the vast array of documentation and other online recourses to produce example programs for debugging.

Relevant: The emulation project aligns with the objectives of exploring and understanding various hardware development techniques and methodologies. Relevant in historical, educational and hobbyist contexts. The project will be used in a potential future project exploring hardware verification techniques relevant to various industries including embedded system design and hardware testing.

Time-bound: The artefact will be designed and tested according to the following Gantt chart.

Table. 1. Gantt Chart



8 Requirements and Analysis.

Before detailing the design requirements of this FPGA, it is important to understand the different steps required to execute a single 6502 instruction.

This segment will execute the example instruction of ADC, abs.

8.1 Instruction Format

6502 instructions are comprised of an Instruction and an Addressing Mode. The Instruction determines the operation in this case ADC means add. Then the Addressing Mode indicates the value to be used in this case the value is pointed to by an absolute.

It is common practice to represent the Instruction as a 3 uppercase letter mnemonic followed by an Addressing mode.

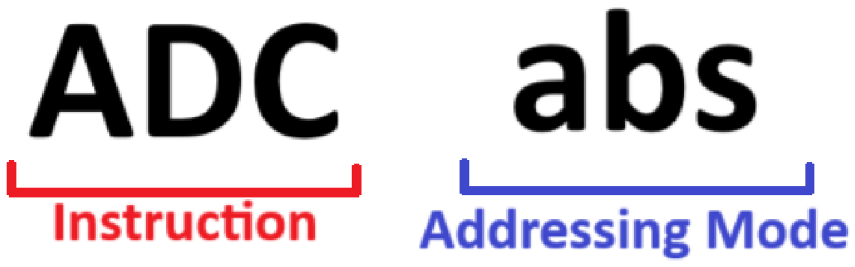


Fig. 1. Example Instruction and Addressing Mode.

8.2 Clock Cycle.

To complete an instruction the FPGA must move between three possible states, Fetch, Decode and Execute with a Clock Cycle completing after the FPGA has been in each state once.

Exactly what happens in each state changes depends on which step of which instruction is currently being processed, for example if multiple clock cycles are required to retrieve a piece of information critical to the execution of the instruction. The FPGA will do nothing during its execution state and wait until it has finished decoding.

To accomplish this a flag is raised to indicate if the FPGA wants to do something in the next state. So, after the critical information is retrieved and has been retrieved during Decode state, the FPGA will raise an “Execution” flag to indicate it should do something during the next execution state.

8.3 Registers.

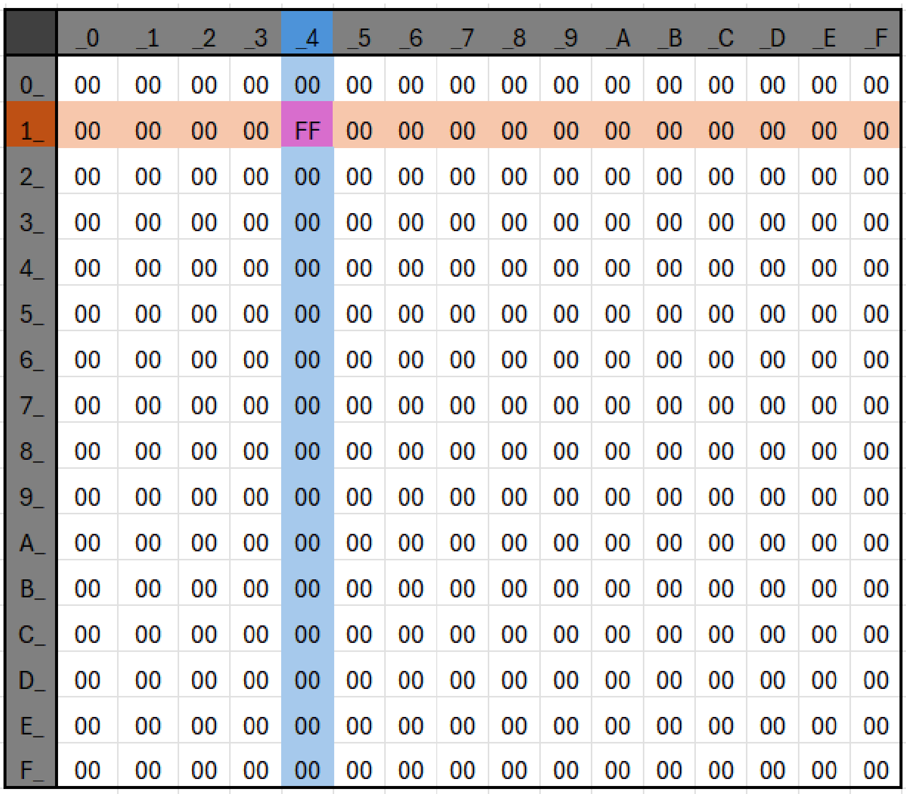
The 6502 has eight Registers. Each register can store a value but only four will be required for this explanation.

- **Program Counter**. will record the current location in memory that we want to look at.
- **Current Instruction Register**. will be used to store the instruction and addressing mode once it has been fetched.
- **Memory Buffer Register**. will store any memory addresses we might need to use to find the value pointed at by the addressing mode.
- **Accumulator.** is the location where the value having the instruction operated on it is stored. In the example whatever value is having a number added to it is stored in the accumulator and after the instruction has been executed this is where the result will be stored.

8.4 Memory.

The 6502 has 2^16 or 65536 Bytes of memory. Each memory location has an associated value called its memory location and a stored value called its memory value. When reading from memory the FPGA must go to a specific memory location and retrieve its memory value. The 6502 is an 8bit processor therefore each memory value has a range between 0-255 or 00-FF in Hexadecimal. For example, if we wanted to read the memory value of memory location 20 (writ-ten as 14 in Hexadecimal). We can go to this location and retrieve a value of FF or 255.

It is important to note that an 8-bit value can only represent 256 bytes of memory. To access any higher valued memory addresses a larger 16bit number is needed. Because of this the first 256 memory are in the zero page as only an 8-bit number is required to find it. To find any other location the highbyte or first 8-bits of the 16-bit address is used to find the page number and the lowbyte is used to find the address. For example, the memory location 0114 has a highbyte of 01 and a lowbyte of 14 and is therefore at location 14 on page 01. A visual representation of memory retrieval can be seen in Figure. 3.



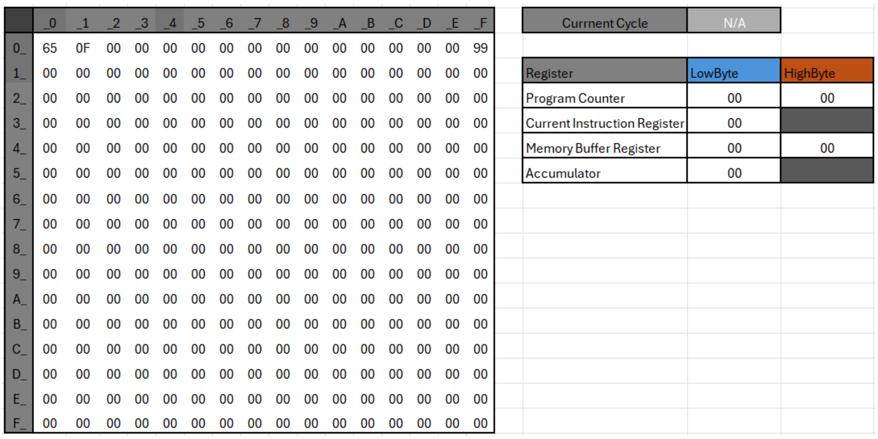
	_0	_1	_2	_3	_4	_5	_6	_7	_8	_9	_A	_B	_C	_D	_E	_F
0_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
1_	00	00	00	00	FF	00	00	00	00	00	00	00	00	00	00	00
2_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
3_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
4_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
5_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
6_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
7_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
8_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
9_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
A_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
B_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
C_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
D_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
E_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
F_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Fig. 2. Visual Representation of memory Retrieval.

8.5 Example Execution.

This example starts with all registers empty, and the program loaded into memory. The zero page of memory is represented on the right., The current step of the fetch, decode, execute cycle on the left and the registers are displayed below the state.

Some of the 6502's registers can store a 16-bit value and others can only store 8-bit value. The registers that have an 8-bit limit have their highbyte greyed out. The example begins with Figure. 4.



	_0	_1	_2	_3	_4	_5	_6	_7	_8	_9	_A	_B	_C	_D	_E	_F
0_	65	0F	00	00	00	00	00	00	00	00	00	00	00	00	00	99
1_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
2_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
3_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
4_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
5_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
6_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
7_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
8_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
9_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
A_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
B_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
C_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
D_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
E_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
F_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Current Cycle	N/A

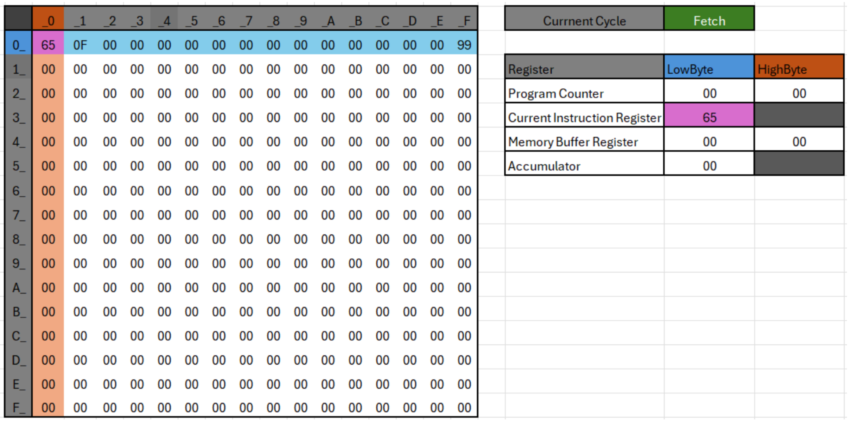
Register	LowByte	HighByte
Program Counter	00	00
Current Instruction Register	00	
Memory Buffer Register	00	00
Accumulator	00	

Fig. 3. Example Instruction 1.

State Fetch. Cycle 0

The clock cycle continues to the fetch state where the value (65), which is stored in memory location 00 on the 00 page (zeropage), as indicated by the Program Counter is loaded into the Current Instruction Register.

65 is the value used to represent ADC, abs during this step this value is passed through the decoder to determine the steps required to perform this instruction. the addressing mode “abs” requires something to happen on the decode step so the decode flag is raised allowing the FPGA to do something during the next cycle.



	_0	_1	_2	_3	_4	_5	_6	_7	_8	_9	_A	_B	_C	_D	_E	_F
0_	65	0F	00	00	00	00	00	00	00	00	00	00	00	00	00	99
1_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
2_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
3_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
4_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
5_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
6_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
7_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
8_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
9_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
A_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
B_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
C_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
D_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
E_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
F_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Current Cycle	Fetch

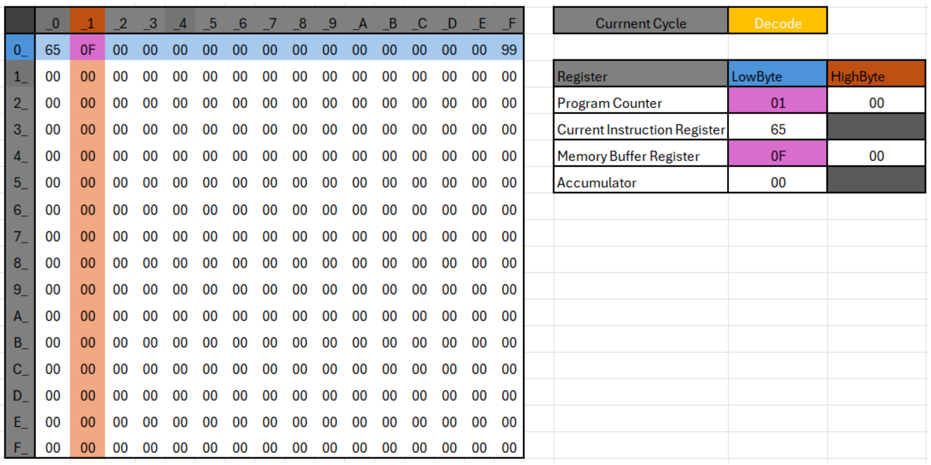
Register	LowByte	HighByte
Program Counter	00	00
Current Instruction Register	65	
Memory Buffer Register	00	00
Accumulator	00	

Fig. 4. Example Instruction 2.

State Decode. Cycle 0

The addressing mode abs, requires a value so during the decode step the FPGA will increment the program counter and copy the associated memory address into the lowbyte of the Memory Buffer Register.

However, the Addressing Mode requires an absolute address or a 16-bit address. This means that the execute flag is not raised resulting in nothing happening during the following execute and fetch step.



	_0	_1	_2	_3	_4	_5	_6	_7	_8	_9	_A	_B	_C	_D	_E	_F
0_	65	0F	00	00	00	00	00	00	00	00	00	00	00	00	00	99
1_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
2_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
3_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
4_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
5_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
6_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
7_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
8_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
9_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
A_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
B_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
C_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
D_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
E_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
F_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

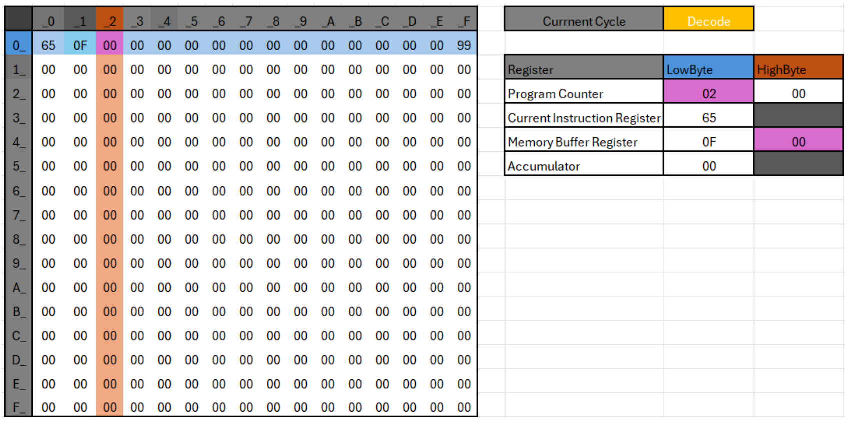
Current Cycle	Decode

Register	LowByte	HighByte
Program Counter	01	00
Current Instruction Register	65	
Memory Buffer Register	0F	00
Accumulator	00	

Fig. 5. Example Execution 3.

State Decode. Cycle 1

During the next decode state the program counter is again incremented. This time the referenced value is loaded into the Highbyte. The FPGA now has a complete 16-bit address, 000F, the Addressing Mode abs, states the value needed is at the memory location pointed at by the Memory Buffer Register. The execute flag is then raised.



	_0	_1	_2	_3	_4	_5	_6	_7	_8	_9	_A	_B	_C	_D	_E	_F
0_	65	0F	00	00	00	00	00	00	00	00	00	00	00	00	00	99
1_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
2_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
3_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
4_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
5_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
6_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
7_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
8_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
9_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
A_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
B_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
C_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
D_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
E_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
F_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

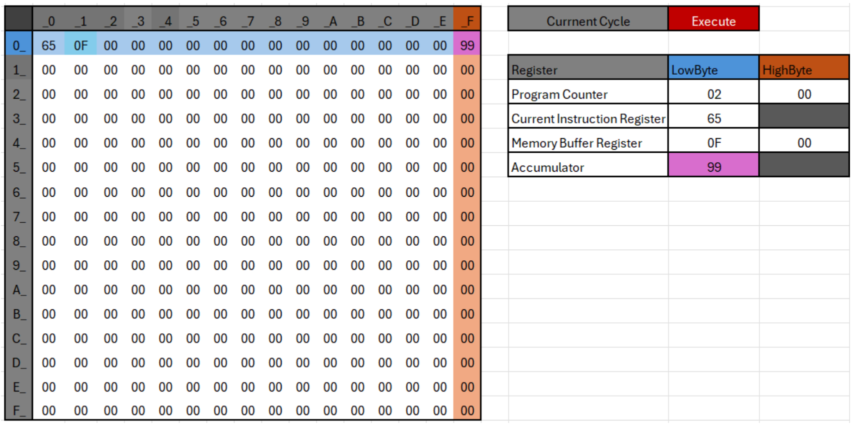
Currnent Cycle	Decode

Register	LowByte	HighByte
Program Counter	02	00
Current Instruction Register	65	
Memory Buffer Register	0F	00
Accumulator	00	

Fig. 6. Example Execution 4.

State Execute. Cycle 1

Finally, the instruction is executed by adding the value at the indicated memory location to the accumulator in this case 99 (153 in Denary) to 00 resulting with 99 being stored in the accumulator. The Fetch Flag would then be raised as the current instruction has been executed and the FPGA is ready to fetch the next one. This concludes the example execution.



	_0	_1	_2	_3	_4	_5	_6	_7	_8	_9	_A	_B	_C	_D	_E	_F
0_	65	0F	00	00	00	00	00	00	00	00	00	00	00	00	00	99
1_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
2_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
3_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
4_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
5_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
6_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
7_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
8_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
9_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
A_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
B_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
C_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
D_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
E_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
F_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Currnent Cycle	Execute

Register	LowByte	HighByte
Program Counter	02	00
Current Instruction Register	65	
Memory Buffer Register	0F	00
Accumulator	99	

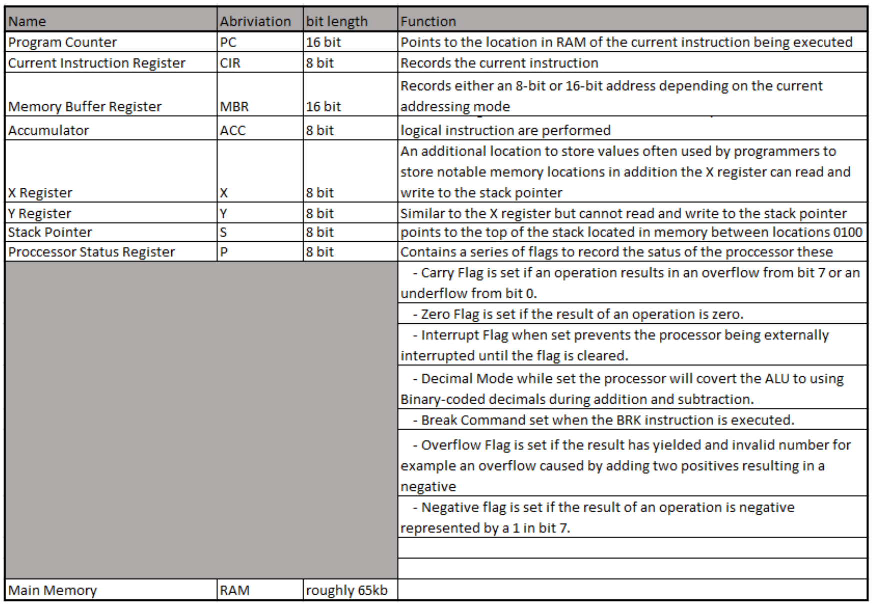
Fig. 7. Example Execution 5.

8.6 Instruction Set Overview.

The following graphs show all the registers, instructions and addressing modes required to emulate 6502.

Table. 2. is a list of all registers accessible to a programmer the bit length and function columns describe their bit length and use.

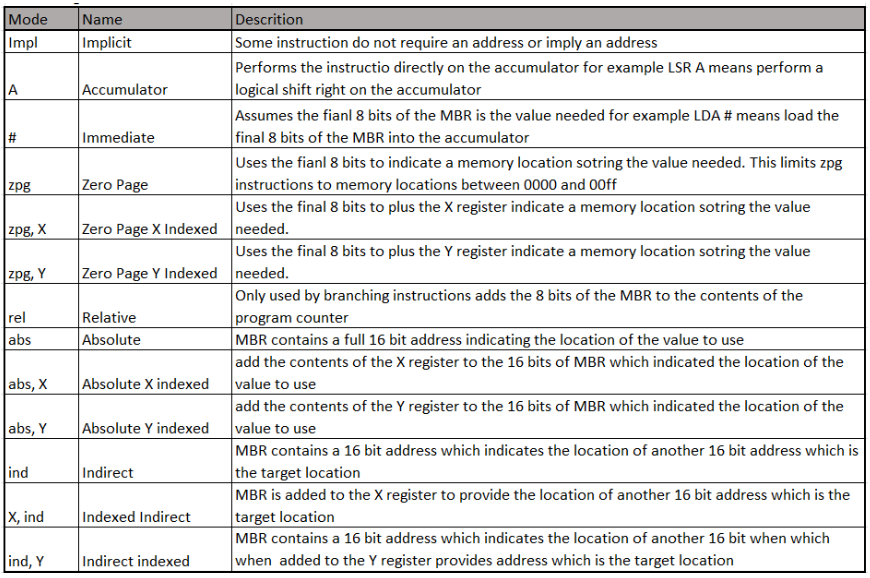
Table. 2. 6502 register list.



Name	Abriviation	bit length	Function
Program Counter	PC	16 bit	Points to the location in RAM of the current instruction being executed
Current Instruction Register	CIR	8 bit	Records the current instruction
Memory Buffer Register	MBR	16 bit	Records either an 8-bit or 16-bit address depending on the current addressing mode
Accumulator	ACC	8 bit	logical instruction are performed
X Register	X	8 bit	An additional location to store values often used by programmers to store notable memory locations in addition the X register can read and write to the stack pointer
Y Register	Y	8 bit	Similar to the X register but cannot read and write to the stack pointer
Stack Pointer	S	8 bit	points to the top of the stack located in memory between locations 0100
Proccessor Status Register	P	8 bit	Contains a series of flags to record the satus of the proccessor these
			- Carry Flag is set if an operation results in an overflow from bit 7 or an underflow from bit 0.
			- Zero Flag is set if the result of an operation is zero.
			- Interrupt Flag when set prevents the processor being externally interrupted until the flag is cleared.
			- Decimal Mode while set the processor will covert the ALU to using Binary-coded decimals during addition and subtraction.
			- Break Command set when the BRK instruction is executed.
			- Overflow Flag is set if the result has yielded and invalid number for example an overflow caused by adding two positives resulting in a negative
			- Negative flag is set if the result of an operation is negative represented by a 1 in bit 7.
Main Memory	RAM	roughly 65kb	

Table. 3. Shows the complete list of addressing modes, note not all combinations of instructions and addressing modes are valid, for example the ADC instruction cannot have an implied addressing mode.

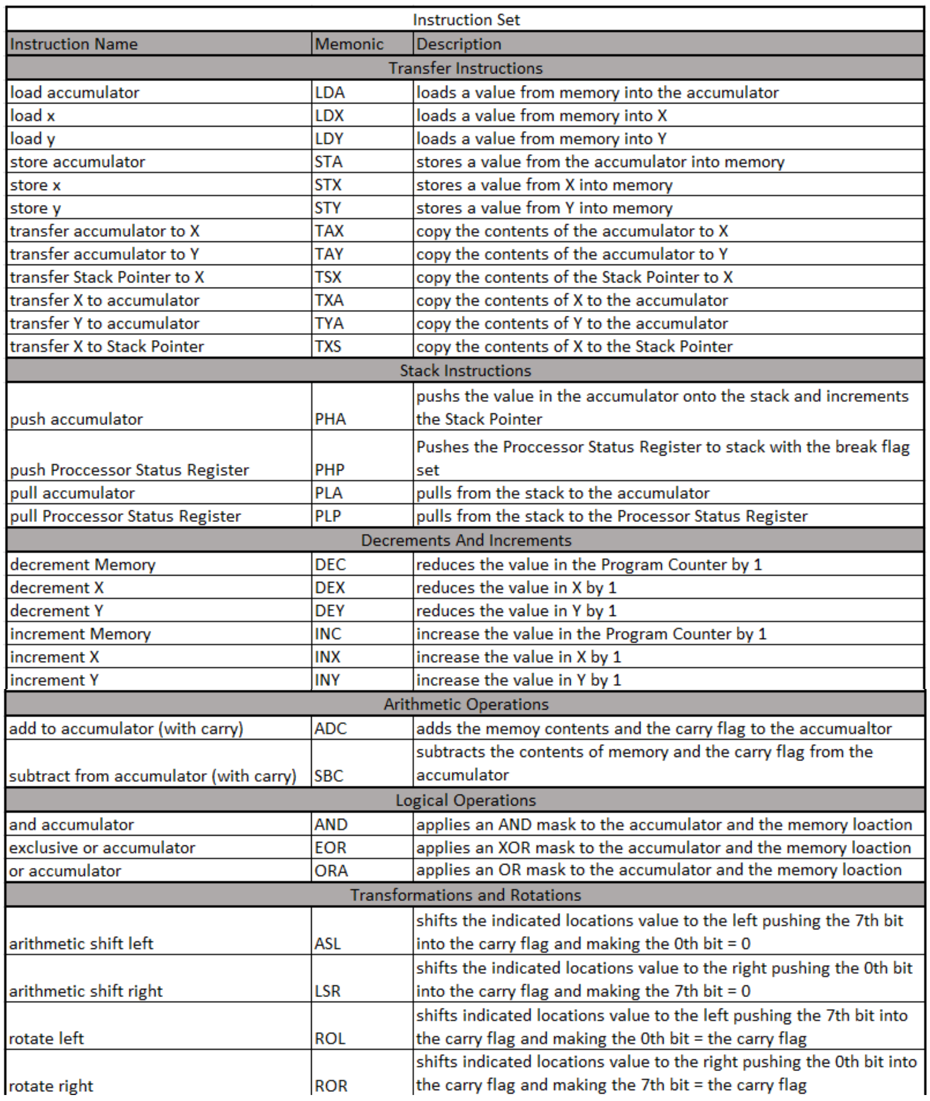
Table. 3. 6502 addressing modes.



Mode	Name	Descrition
Impl	Implicit	Some instruction do not require an address or imply an address
A	Accumulator	Performs the instructio directly on the accumulator for example LSR A means perform a logical shift right on the accumulator
#	Immediate	Assumes the fianl 8 bits of the MBR is the value needed for example LDA # means load the final 8 bits of the MBR into the accumulator
zpg	Zero Page	Uses the fianl 8 bits to indicate a memory location sotring the value needed. This limits zpg instructions to memory locations between 0000 and 00ff
zpg, X	Zero Page X Indexed	Uses the final 8 bits to plus the X register indicate a memory location sotring the value needed.
zpg, Y	Zero Page Y Indexed	Uses the final 8 bits to plus the Y register indicate a memory location sotring the value needed.
rel	Relative	Only used by branching instructions adds the 8 bits of the MBR to the contents of the program counter
abs	Absolute	MBR contains a full 16 bit address indicating the location of the value to use
abs, X	Absolute X indexed	add the contents of the X register to the 16 bits of MBR which indicated the location of the value to use
abs, Y	Absolute Y indexed	add the contents of the Y register to the 16 bits of MBR which indicated the location of the value to use
ind	Indirect	MBR contains a 16 bit address which indicates the location of another 16 bit address which is the target location
X, ind	Indexed Indirect	MBR is added to the X register to provide the location of another 16 bit address which is the target location
ind, Y	Indirect indexed	MBR contains a 16 bit address which indicates the location of another 16 bit when which when added to the Y register provides address which is the target location

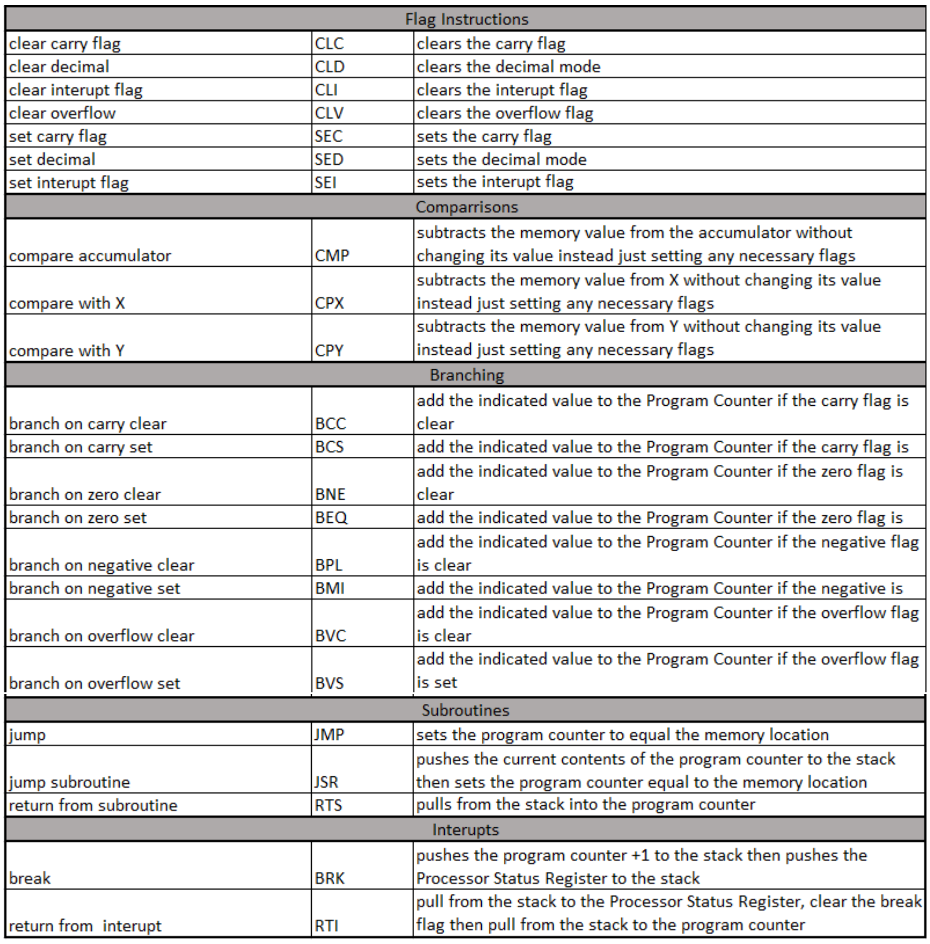
Next Table. 4. and 5. Show the 6502 instruction set. Additional dormant "invalid" instructions do exist, however are not initially accessible to users and will not be implemented as they are outside the scope of this project.

Table. 4. 6502 instruction set part 1.



Instruction Set		
Instruction Name	Memonic	Description
Transfer Instructions		
load accumulator	LDA	loads a value from memory into the accumulator
load x	LDX	loads a value from memory into X
load y	LDY	loads a value from memory into Y
store accumulator	STA	stores a value from the accumulator into memory
store x	STX	stores a value from X into memory
store y	STY	stores a value from Y into memory
transfer accumulator to X	TAX	copy the contents of the accumulator to X
transfer accumulator to Y	TAY	copy the contents of the accumulator to Y
transfer Stack Pointer to X	TSX	copy the contents of the Stack Pointer to X
transfer X to accumulator	TXA	copy the contents of X to the accumulator
transfer Y to accumulator	TYA	copy the contents of Y to the accumulator
transfer X to Stack Pointer	TXS	copy the contents of X to the Stack Pointer
Stack Instructions		
push accumulator	PHA	pushs the value in the accumulator onto the stack and increments the Stack Pointer
push Proccessor Status Register	PHP	Pushes the Proccessor Status Register to stack with the break flag set
pull accumulator	PLA	pulls from the stack to the accumulator
pull Proccessor Status Register	PLP	pulls from the stack to the Processor Status Register
Decrements And Increments		
decrement Memory	DEC	reduces the value in the Program Counter by 1
decrement X	DEX	reduces the value in X by 1
decrement Y	DEY	reduces the value in Y by 1
increment Memory	INC	increase the value in the Program Counter by 1
increment X	INX	increase the value in X by 1
increment Y	INY	increase the value in Y by 1
Arithmetic Operations		
add to accumulator (with carry)	ADC	adds the memoy contents and the carry flag to the accumualtor
subtract from accumulator (with carry)	SBC	subtracts the contents of memory and the carry flag from the accumulator
Logical Operations		
and accumulator	AND	applies an AND mask to the accumulator and the memory loaction
exclusive or accumulator	EOR	applies an XOR mask to the accumulator and the memory loaction
or accumulator	ORA	applies an OR mask to the accumulator and the memory loaction
Transformations and Rotations		
arithmetic shift left	ASL	shifts the indicated locations value to the left pushing the 7th bit into the carry flag and making the 0th bit = 0
arithmetic shift right	LSR	shifts the indicated locations value to the right pushing the 0th bit into the carry flag and making the 7th bit = 0
rotate left	ROL	shifts indicated locations value to the left pushing the 7th bit into the carry flag and making the 0th bit = the carry flag
rotate right	ROR	shifts indicated locations value to the right pushing the 0th bit into the carry flag and making the 7th bit = the carry flag

Table. 5. 6502 instruction set part 2.



Flag Instructions		
clear carry flag	CLC	clears the carry flag
clear decimal	CLD	clears the decimal mode
clear interupt flag	CLI	clears the interupt flag
clear overflow	CLV	clears the overflow flag
set carry flag	SEC	sets the carry flag
set decimal	SED	sets the decimal mode
set interupt flag	SEI	sets the interupt flag
Comparrisons		
compare accumulator	CMP	subtracts the memory value from the accumulator without changing its value instead just setting any necessary flags
compare with X	CPX	subtracts the memory value from X without changing its value instead just setting any necessary flags
compare with Y	CPY	subtracts the memory value from Y without changing its value instead just setting any necessary flags
Branching		
branch on carry clear	BCC	add the indicated value to the Program Counter if the carry flag is clear
branch on carry set	BCS	add the indicated value to the Program Counter if the carry flag is
branch on zero clear	BNE	add the indicated value to the Program Counter if the zero flag is clear
branch on zero set	BEQ	add the indicated value to the Program Counter if the zero flag is
branch on negative clear	BPL	add the indicated value to the Program Counter if the negative flag is clear
branch on negative set	BMI	add the indicated value to the Program Counter if the negative is
branch on overflow clear	BVC	add the indicated value to the Program Counter if the overflow flag is clear
branch on overflow set	BVS	add the indicated value to the Program Counter if the overflow flag is set
Subroutines		
jump	JMP	sets the program counter to equal the memory location
jump subroutine	JSR	pushes the current contents of the program counter to the stack then sets the program counter equal to the memory location
return from subroutine	RTS	pulls from the stack into the program counter
Interupts		
break	BRK	pushes the program counter +1 to the stack then pushes the Processor Status Register to the stack
return from interupt	RTI	pull from the stack to the Processor Status Register, clear the break flag then pull from the stack to the program counter

Finally, all valid combinations of instructions and addressing mode can be written as an 8-bit number that can be read into the CIR during the fetch state. Table. 6. is a matrix that shows the instruction and addressing mode for each number. For example, ADC, abs can be found on column 6 and row 5 therefore making its 8-bit number 65.

Table. 6. 6502 instruction matrices.

	_0	_1	_2	_3	_4	_5	_6	_7	_8	_9	_a	_b	_c	_d	_e	_f
0_	BRK impl	ORA X, ind				ORA zpg	ASL zpg		PHP impl	ORA #	ASL A			ORA abs	ASL abs	
1_	BPL rel	ORA ind, Y				ORA zpg, X	ASL zpg, X		CLC impl	ORA abs, Y				ORA abs, X	ASL abs, X	
2_	JSR abs	AND X, ind				AND zpg	ROL zpg		PLP impl	AND #	ROL A			AND abs	ROL abs	
3_	BMI rel	AND ind, Y				AND zpg, X	ROL zpg, X		SEC impl	AND abs, Y				AND abs, X	ROL abs, X	
4_	RTI impl	EOR X, ind				EOR zpg	LSR zpg		PHA impl	EOR #	LSR A		JMP abs	EOR abs	LSR abs	
5_	BVC rel	EOR ind, Y				EOR zpg, X	LSR zpg, X		CLI impl	EOR abs, Y				EOR abs, X	LSR abs, X	
6_	RTS impl	ADC X, ind				ADC zpg	ROR zpg		PLA impl	ADC #	ROR A		JMP ind	ADC abs	ROR abs	
7_	BVS rel	ADC ind, Y				ADC zpg, X	ROR zpg, X		SEI impl	ADC abs, Y				ADC abs, X	ROR abs, X	
8_		STA X, ind			STY zpg	STA zpg	STX zpg		DEY impl		TXA impl		STY abs	STA abs	STX abs	
9_	BCC rel	STA ind, Y			STY zpg, X	STA zpg, X	STX zpg, Y		TYA impl	STA abs, Y	TXS impl			STA abs, X		
a_	LDY #	LDA X, ind	LDX #		LDY zpg	LDA zpg	LDX zpg		TAY impl	LDA #	TAX impl		LDY abs	LDA abs	LDX abs	
b_	BCS rel	LDA ind, Y			LDY zpg, X	LDA zpg, X	LDX zpg, Y		CLV imp	LDA abs, Y	TSX impl		LDY abs, X	LDA abs, X	LDX abs, Y	
c_	CPY #	CMP X, ind			CPY zpg	CMP zpg	DEC zpg		INY impl	CMP #	DEX impl		CPY abs	CMP abs	DEC abs	
d_	BNE rel	CMP ind, Y				CMP zpg, X	DEC zpg, X		CLD impl	CMP abs, Y				CMP abs, X	DEC abs, X	
e_	CPX #	SBC X, ind			CPX zpg	SBC zpg	INC zpg		INX impl	SBC #			CPX abs	SBC abs	INC abs	
f_	BEQ rel	SBC ind, Y				SBC zpg, X	INC zpg, X		SED impl	SBC abs, Y				SBC abs, X	INC abs, X	

To fulfill the aims and objectives of this project must be able to execute all combinations of instructions and addressing modes listed in Figure. 11 by cycling through a finite number of clock states. If done successfully users will be able to use the FPGA to run any program written in 6502 machine code.

9 Requirements and Analysis.

Software development Methodology: This project will adopt a waterfall development methodology. In FPGA design, the typical approach begins with the initial clock state and progresses to the final one. This sequencing is deliberate, as the first clock state operates independently of any other state, facilitating isolated debugging. However, if a bug is discovered in an earlier state during the implementation of subsequent states, necessitating a redesign, all subsequent states must also undergo revision. Consequently, this project mandates the sequential design, testing, and implementation of foundational circuits before progressing to the subsequent circuit.

Toolset and Machine Environments: This project will use Logisim Evolution for three key reasons. Firstly, its visual and intuitive user interface enhances readability, enabling readers to follow the development process more easily. Secondly, I have enough experience using this hardware development environment to undertake a project like this. Finally, Logisim Evolution allows storing designs in both .circ and .vhd formats, ensuring compatibility for users proficient in either Logisim Evolution or VHDL.

Design: This project will use prototypes to both design and explain complex circuits, hopefully preventing as many errors as possible and allowing readers to better understand the development process.

Testing: This project will employ fixed unit testing to guarantee the intended functionality of prior circuits without compromising subsequent components. For components with no memory elements, test vectors will be written and compared against their outputs to ascertain consistent performance for any given input. However, components integrating memory elements necessitate a different testing approach, as inputs may yield diverse outputs across iterations. To validate such circuits, test programs/ inputs will be written to ensure reliability.

10 Implementation.

10.1 Initial Components

Before designing any complex components that might be used during one of the fetch, decode or execute states. A few initial components must be designed, used during multiple or all clock states. This includes: The Clock, Main Memory, The Program Counter, and some buses.

Clock. The purpose of the clock is to dictate which state of the clock cycle the FPGA is in with different wires signaling to a variety of components when they should activate.

The architecture of a clock progresses from the initial design shown in Figure. 8. This circuit will activate its output pin after three instances of the input pin being turned on. This is accomplished by linking D-type flip flops together.

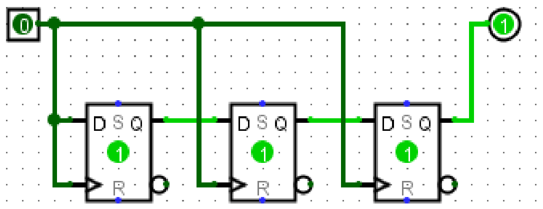


Fig. 8. Clock Component 1.

However, a clock needs to have multiple output pins activating at different times. To comply with this requirement D-type flip flops must turn off after their successor is activated. If multiple flip flops are active, it would imply that the FPGA is in multiple states. To accomplish this a NOT of the next D-type flip flop is fed into an AND gate preventing each flip flop from reading a one while its successor is active.

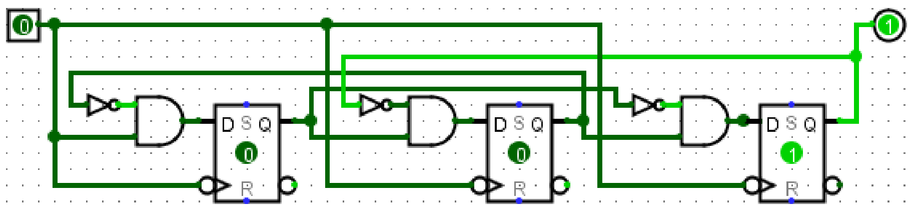


Fig. 9. Clock Component 2.

Now the clock will have only one active D-type flip flop at a time. However, due to the first D-type flip flop being initially activated by an input pin it can lead to states such as in Figure. 9. where the initial flip flop reactivates during another state.

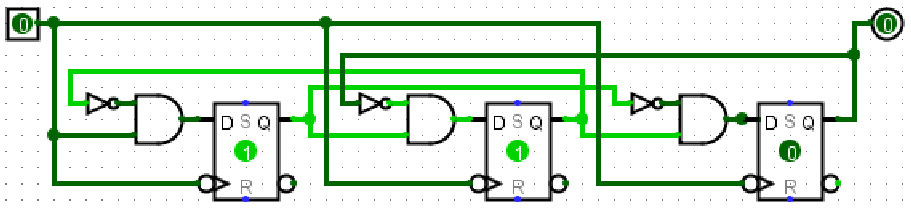


Fig. 10. Clock Component 3.

The simplest method to resolve this is to make the input of the initial flip flop NOT of every other flip flop. This simplifies the design, halving the number of NOT gates required and prevents the clock signaling multiple states at once.

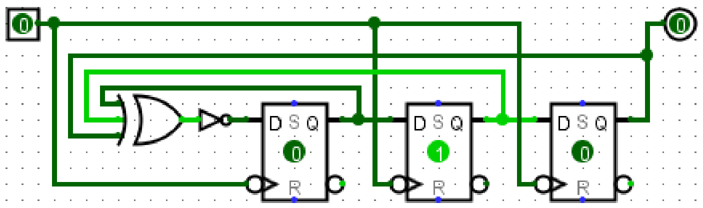


Fig. 11. Clock Component 4.

The final design can be seen in Figure. 11. Four D-type flip flops are used to represent 4 potential states that the clock can be in Fetch, Decode (Lowbyte), Decode (Highbyte) and Execute. I elected to add an additional state to allow the FPGA to decode a 16-bit address in a single clock cycle; thinking this would reduce the number of ticks required to complete an instruction requiring an absolute address. The efficiency impact of this decision and my FPGA will be discussed later in this document.

Additionally, AND gates have been added to the output pins for the different states of the clock to prevent “Flicker”. While electrical signals appear instant a small amount of time is required for signals to travel down wires and some components down particularly long wires may still receive an activation signal for a brief period at the beginning of the next state, usually resulting in data collisions on the bus. The AND gates cause the clock to transmit no signal before changing to the next state.

Finally, an additional output pin for the raw clock signal has been added which will be used primarily by memory devices like registers to synchronize across the FPGA.

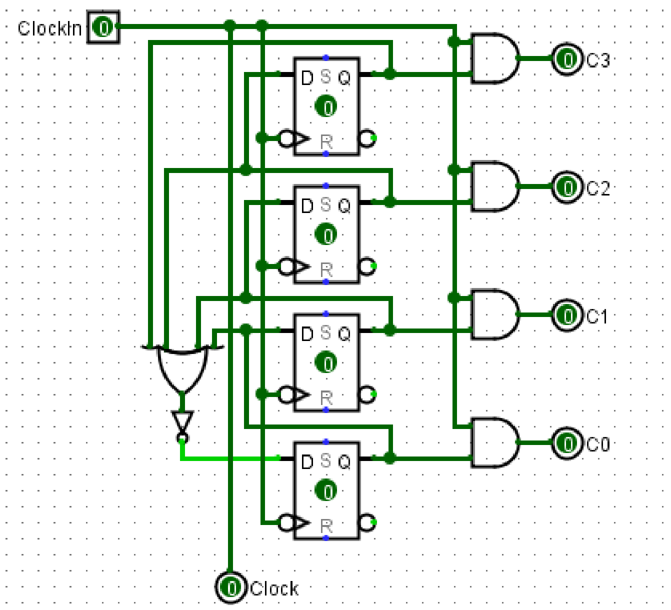


Fig. 12. Final clock component design.

Program Counter. The fundamental design of a Program Counter is a register connected to an adder that inputs back into the register seen in Figure. 12. Every time the input pin is activated the contents of the register are incremented by one.

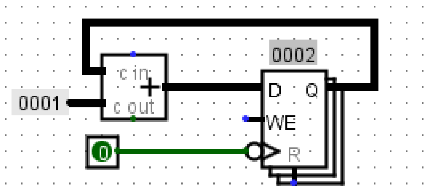


Fig. 13. Program Counter 1.

However, if a branch instruction is executed the contents of the program counter must be overridden. To do this a mechanism must be designed to allow the circuit to alternate the input into the register.

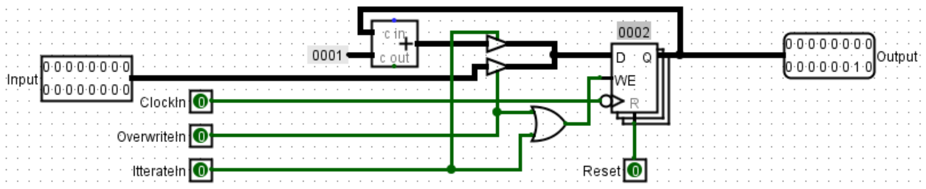


Fig. 14. Final Program Counter.

The final circuit as seen in Figure. 14. Has four input pins that do the following:

- **ClockIn**. Inputs the raw clock signal synchronizing when the register re-writes itself at the end of a valid clock cycle.
- **ItterateIn**. Opens a buffer allowing the register to read its contents +1. Then causes a re-write.
- **OverwriteIn**. Opens a buffer allowing the register to read from the input pin and causes a re-write.
- **Input**. In the event of a branch instruction the input pin is the location the program counter should branch to.
- **Reset**. Clears the contents of the Program Counter, reseting it to 0000.

Memory, Registers, and Initial Architecture. First The Clock Component outputs are passed through an and gate with their respective flag. This is to allow the FPGA to toggle if it does something during a specific state. For example, if an additional decode cycle is required before execution. A later circuit will not raise the ExecuteFlag until the appropriate flag is raised.

Additionally, an extra iterate wire will be activated during the fetch, decode(lowbyte) and decode(highbyte) states because all these states require the PC to iterate.

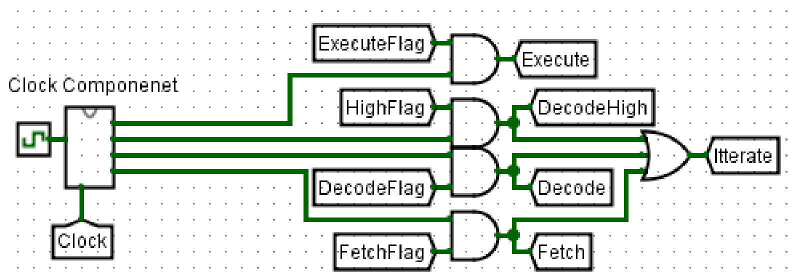


Fig. 15. Clock.

Next the program counter and memory are implemented. The Program Counter is connected to the iterate and clock wires. Additionally, the PCWrite pin, the PCInstOut and Clear wires have been added.

The Memory component used is the default Logisim RAM component. It will read from the memory location indicated by the Program Counter during an iterate step. Additional MWrite and MRead wires have been added for when the FPGA must read or write to memory during the execute state.

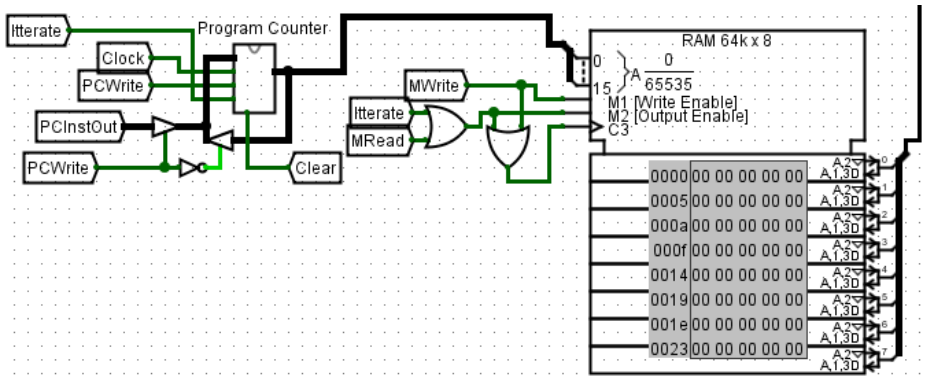


Fig. 16. Program Counter and RAM.

After memory the CIR and MBR registers are implemented. The registers receive their input from RAM with buffers preventing them from reading during an incorrect state. The registers have also been connected to the clear wire.

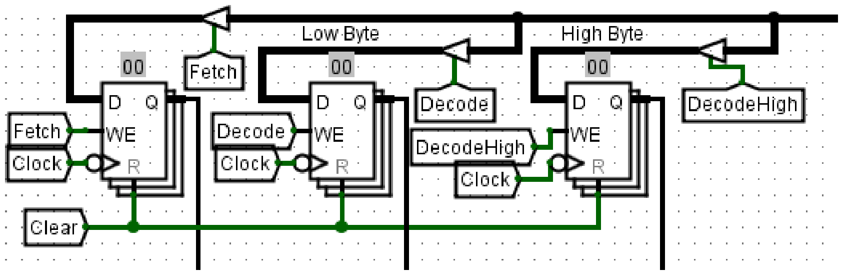


Fig. 17. CIR and MBR.

the current FPGA as seen in Figure. 18. is far from complete, and these components might still receive additions as the structure increases in complexity. However, Figure. 17. will form the backbone of the FPGA and must be tested to ensure it functions correctly.

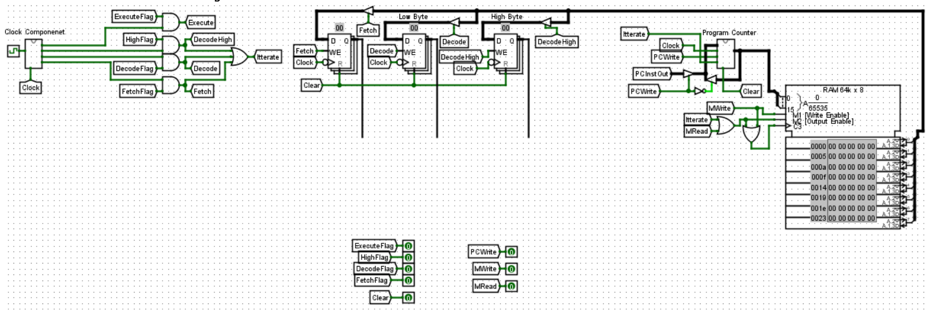


Fig. 18. Initial Architecture.

Currently no mechanism exists to raise clock flags. If the flags are raised manually the FPGA can load values from memory into the CIR and MBR. To test these components work together the memory locations 00, 01 and 02 will be filled with test values and if by the execute state the FPGA has iterated through these values, loading them into the CIR, lowbyte and highbyte we can conclude these components work as intended.

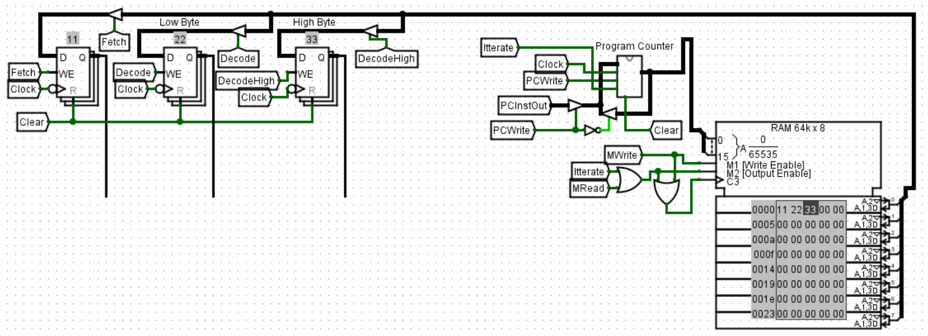


Fig. 19. Loading Example Values into the CIR and MBR across one clock cycle.

10.2 Fetch State.

Fetch is the first and simplest state of the clock cycle. During this state the FPGA must derive an Instruction and Addressing Mode from the contents of the CIR by passing it through a decoder.

Decoder. The function of the decoder is to input an 8-bit number and if it is a valid instruction set a wire to represent the relevant instruction and addressing mode. The circuit required to do this is large but relatively simple.

First an 8-bit number is input and must be split across two 4-bit decoders for two reasons. First the practical limitations, Logisim largest decoder receives 5-bit input allowing for a maximum of 64 instructions whereas the 6502 supports potentially 256 instructions. Secondly if you inspect Table. 6. you will notice that instructions are typically grouped into rows and addressing modes are typically grouped into columns. Building the decoder like this will reduce the total wire length making the component smaller and cheaper to produce.

Figure. 20. shows the initial design. It is supposed to replicate Table. 6. with the top decoder handling the right most 4 bits and the lower decoder handling the left most 4 bits.

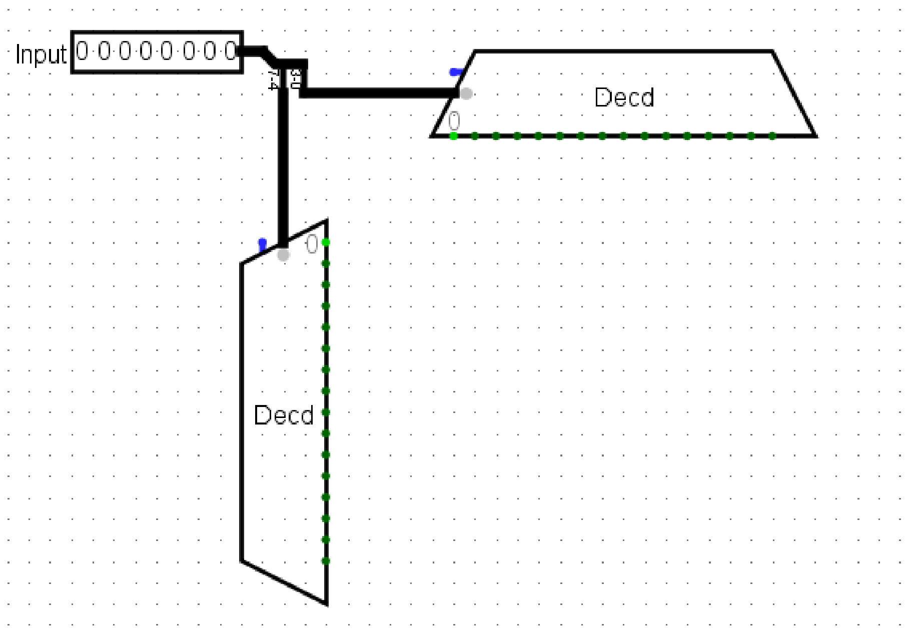


Fig. 20. Decoder 1.

Next the decoder must be able to activate a wire for every single possible input combination. Using AND gates it is possible to determine which input correlates with each instruction.

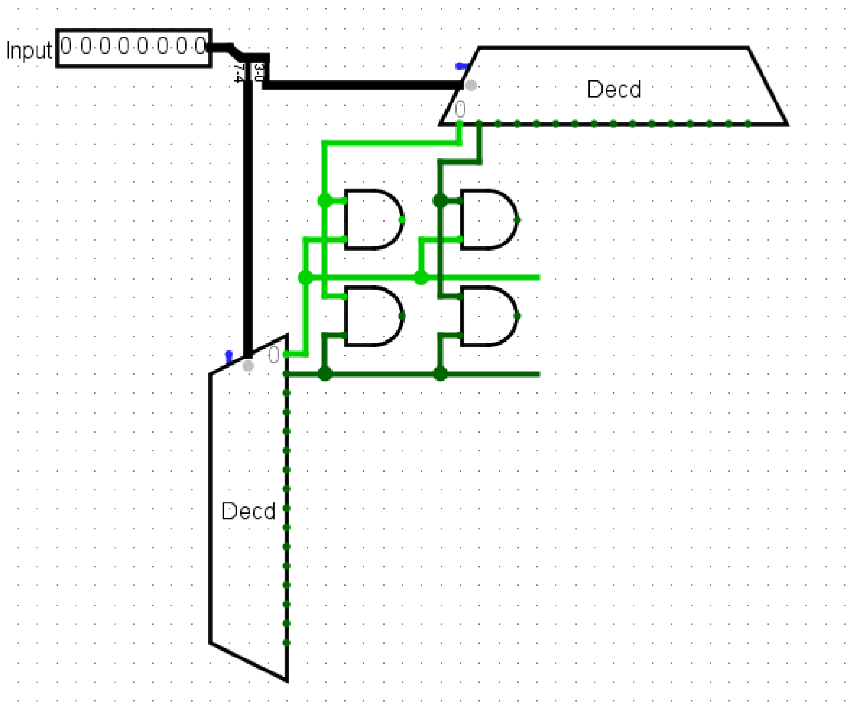


Fig. 21. Decoder 2.

Finally, a method to combine and retrieve identical instructions into a singular wire must be designed. Figure. 22. Assumes all columns share a common addressing mode, and rows share a common instruction. While not accurate this design can be scaled to represent the majority of instructions.

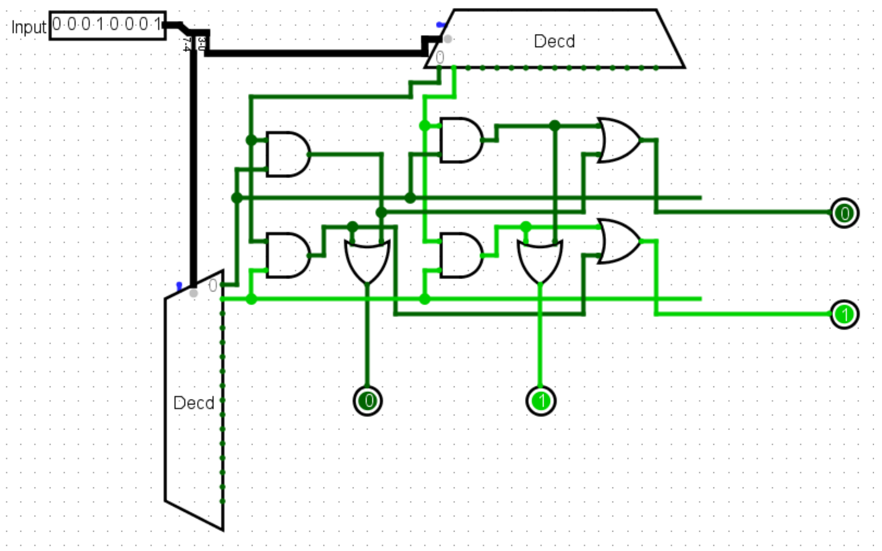


Fig. 22. Decoder 3.

This pattern can be easily repeated to describe the entire instruction set with a few exceptions with the final decoder can be implemented shown in Figure. 23.

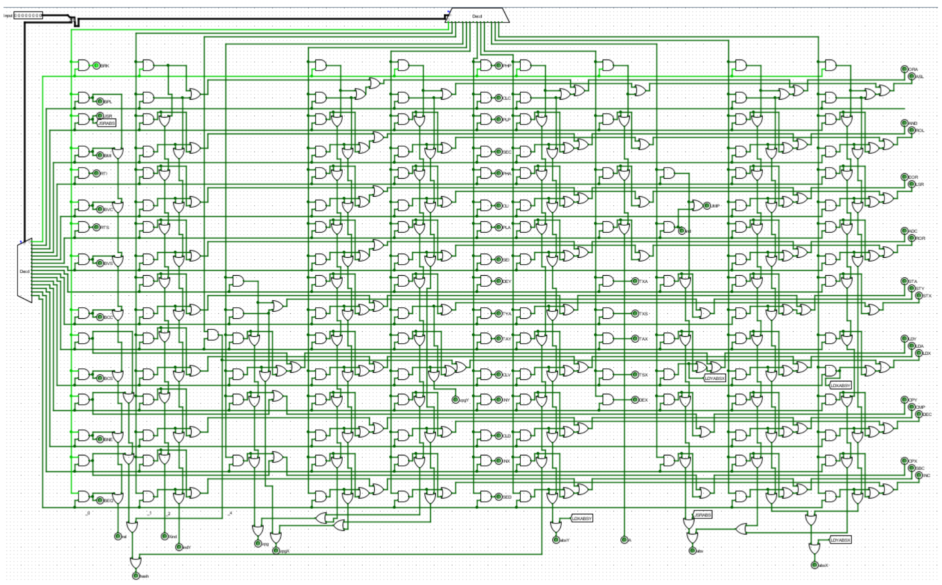
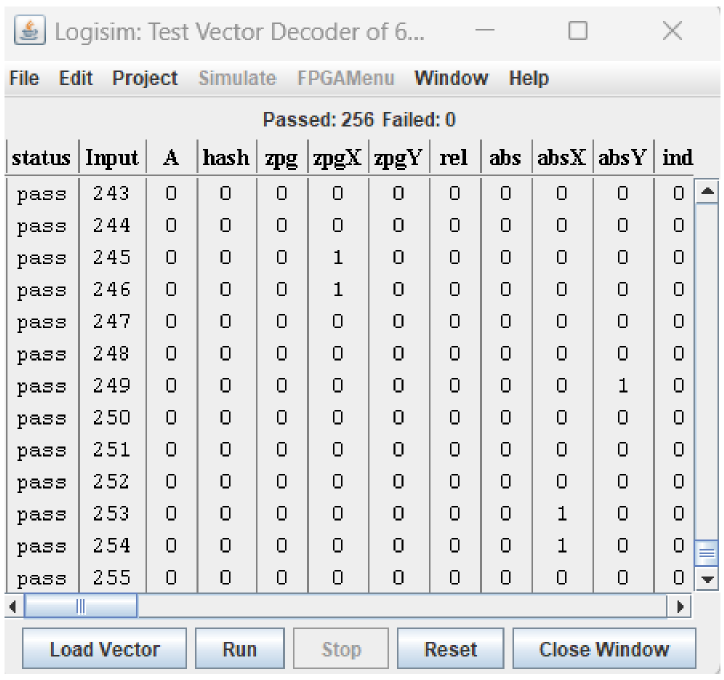


Fig. 23. Final Decoder.

Problems now arise while testing. All previous components have been verifiable manually, but this component is too large to verify manually. Luckily the decoder contains no memory. Meaning the component can be debugged using a test vector. A test vector is a large matrix of possible inputs and expected outputs which can then be compared to confirm that the circuit works as intended.

Logisim has an integrated test vector program which compares expected outputs to actual outputs. It will show how many inputs functioned as intended, how many failed and what the output of failed inputs was. Programs like these are relatively easy to build using a library of your desired language that can feed inputs and read outputs from the component.



Passed: 256 Failed: 0

status	Input	A	hash	zpg	zpgX	zpgY	rel	abs	absX	absY	ind
pass	243	0	0	0	0	0	0	0	0	0	0
pass	244	0	0	0	0	0	0	0	0	0	0
pass	245	0	0	0	1	0	0	0	0	0	0
pass	246	0	0	0	1	0	0	0	0	0	0
pass	247	0	0	0	0	0	0	0	0	0	0
pass	248	0	0	0	0	0	0	0	0	0	0
pass	249	0	0	0	0	0	0	0	0	1	0
pass	250	0	0	0	0	0	0	0	0	0	0
pass	251	0	0	0	0	0	0	0	0	0	0
pass	252	0	0	0	0	0	0	0	0	0	0
pass	253	0	0	0	0	0	0	0	1	0	0
pass	254	0	0	0	0	0	0	0	1	0	0
pass	255	0	0	0	0	0	0	0	0	0	0

Fig. 24. Example execution of the Decoder Test Vector.txt file.

Figure. 24. Shows the test vector being executed passing 256 and failing 0 of the 256 possible input combinations. This method proved effective in finding bugs in the decoder but will likely not be usable for future components using memory components.

10.3 Decode State.

The goal of the Decode state is to retrieve the 8-bit number required for the execution of the current instruction via the addressing mode. So, referring to the example at the beginning to retrieve the number that's going to be added to the accumulator during the execute state via the relevant addressing mode.

To accomplish this a circuit must be built capable of outputting memory locations until the appropriate value is loaded into the lowbyte then signaling when done. This is one of the more complex circuits so for ease of explanation addressing modes will be grouped into:

No Memory Value Required. This includes the impl and A addressing modes which do not require any additional data.

Immediate Memory Required. This includes the rel and # addressing modes which use the immediate next value in memory during execution, requiring one decode(lowbyte) state.

Zeropage Memory Required. This includes the zpg, zpgX and zpgY addressing modes which use the immediate next value in memory to point to a value within the zeropage which is used during execution, requiring two decode(lowbyte)s states.

Absolute Memory Required. This includes the abs, absX and absY addressing modes which use the two immediate next values in memory to point to an absolute 16-bit address which is used during execution, requiring two decode(lowbyte)s and one decode(highbyte)

Indirect Absolute Memory Required. This includes the ind, Xind and indY addressing modes which use the two immediate next values in memory to point to an absolute 16-bit address. Then that memory value and the next memory value are used to create another absolute 16-bit address which points to the value used during execution, requiring 3 decode(lowbytes) and 2 decode(highbytes).

Before designing a circuit to complete the decode state, due to the differing number of decode steps required per addressing mode an initial component must be designed capable of counting how many decode states have happened. Figure. 25. requires the decode pin to be activated three times for the output pin to be activated. Having output pins for each D-type flip flop allows this circuit to count how many decode states have currently happened.

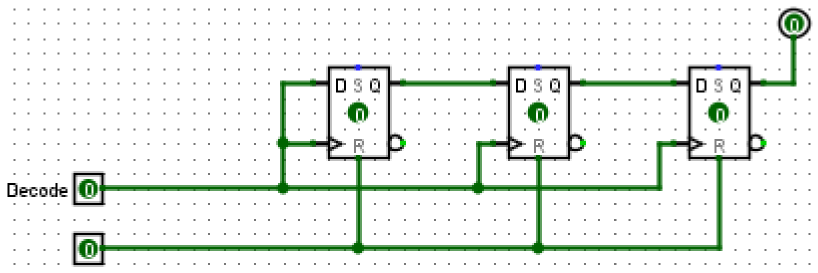


Fig. 25. Input Counter 1.

This next input Counter shown in Figure. 26. uses extra input pins to start from alternating points. Each D-type flip flop will cause the circuit to take one additional decode state before it activates the output pin. So, for example if the second pin which has 2 D-type flip flops ahead of it is activated it will take three decode states to activate the output pin.

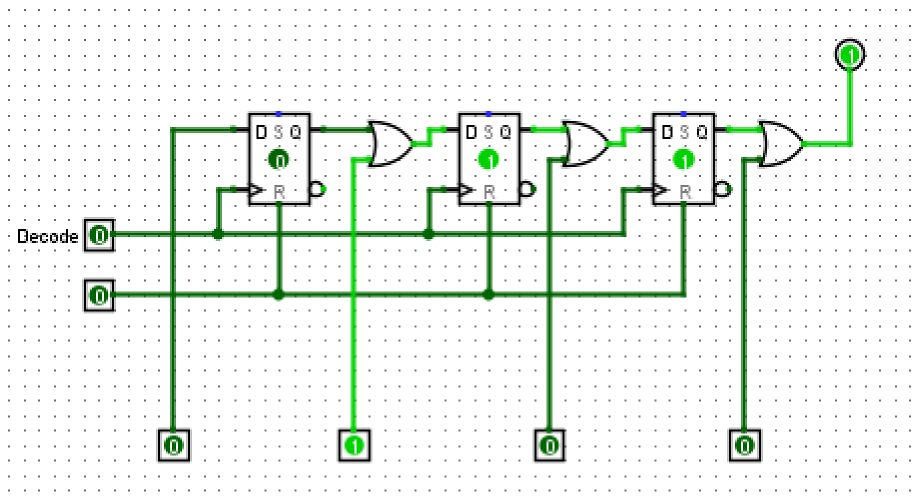


Fig. 26. Input Counter 2.

No Memory Value Required. First the design for the impl and A instruction, shown in Figure. 27. is simple if the circuit doesn't detect any other addressing mode the circuit assumes there must be a No Memory Required Addressing Mode and immediately Raises the execution flag.

Input pins have been added for the X, Y, low and highbyte registers as future addressing modes will require their values.

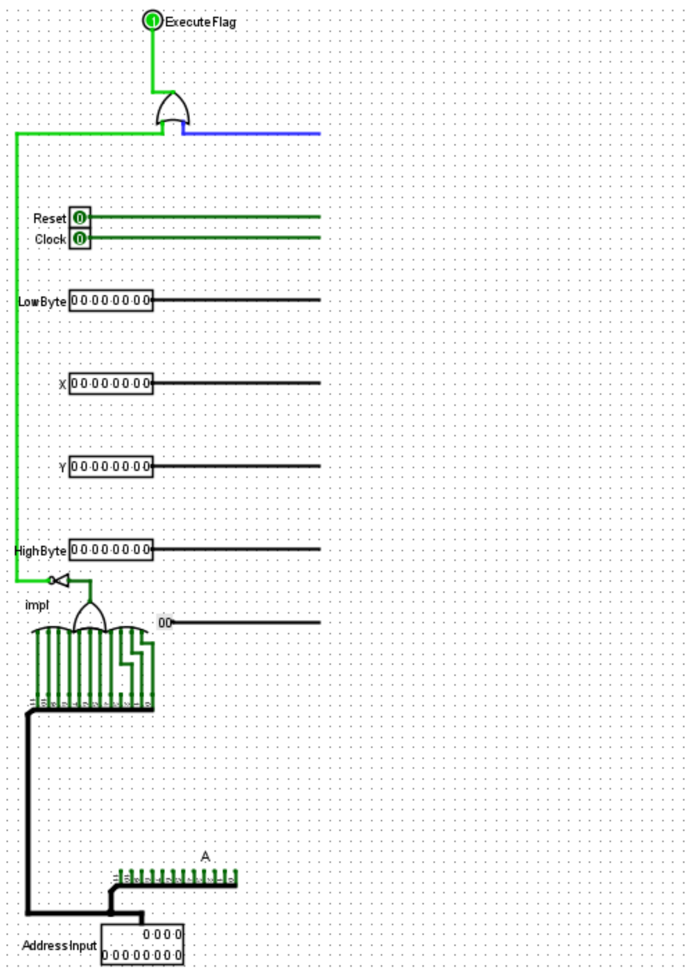


Fig. 27. No Memory Value Required Retriever.

Immediate Memory Required. To achieve an Immediate Memory Addressing Mode one Decode state must be performed therefore when one of these addressing modes is detected the first decode state will activate the first D-type flip flop requiring one decode input before the execute flag is ready.

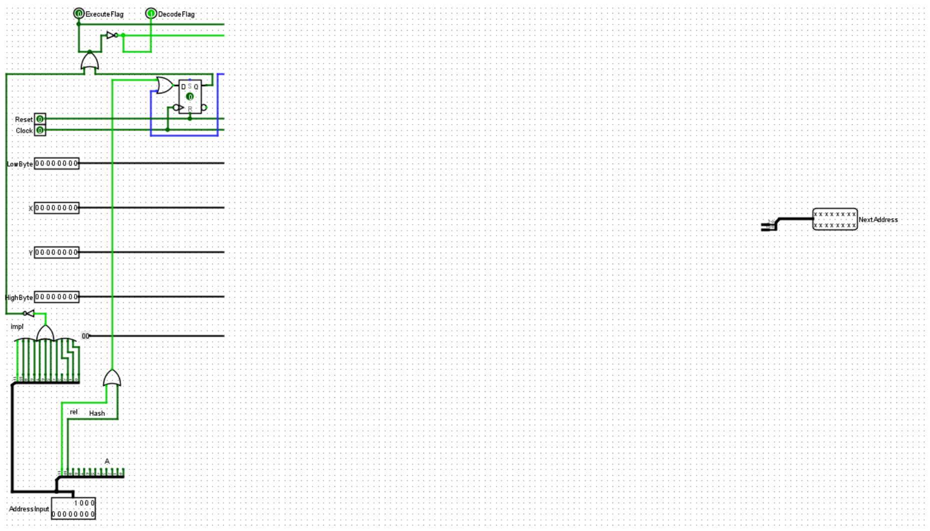


Fig. 28. Immediate Memory Required.

An additional output pin has been added this value will be the memory location that must be read next, however because the Immediate Memory Addressing Mode requires the next immediate memory location there is no need to update the program counter as it already contains the value needed.

Zeropage Memory Required. Next an additional D Type Flip Flop is required for Zeropage Memory Addressing Modes to cause the extra decode state required. Additionally, a new output pin is added called PCoff. This is needed to indicate the Program Counter should not increment. The method the FPGA handles addresses not indicated by the program counter will be explained later in this document.

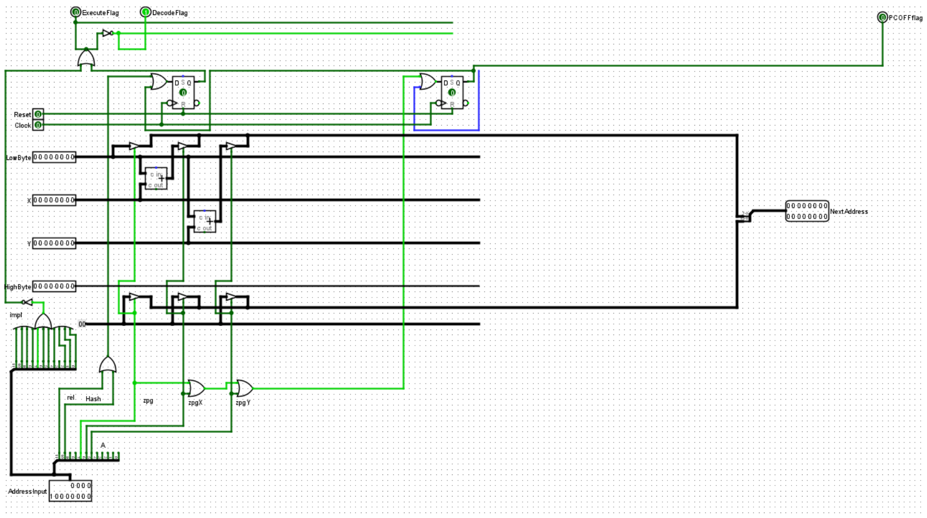


Fig. 29. Zeropage Memory Required.

Absolute Memory Required. Implementing Absolute Memory Addressing Modes is very similar to the previous Addressing Modes. The main difference is now we must worry about decoding the highbyte.

First a HighbyteFlag is added this is to cause something to happen during the de-cocde(highbyte) state. Secondly, we encounter a problem when turning the PC off. If the PC is turned off at the end of the decode(lowbyte) state this causes the de-code(highbyte) state to not iterate the program counter, resulting in the next Fetch State to be one memory location behind the memory location that should be being retrieved. To fix this an additional D Type Flip Flop is added to record when the first decode(highbyte) has ended only then turning the PC off. This Flip Flop is only used for Absolute and Indexed Memory Addressing Modes.

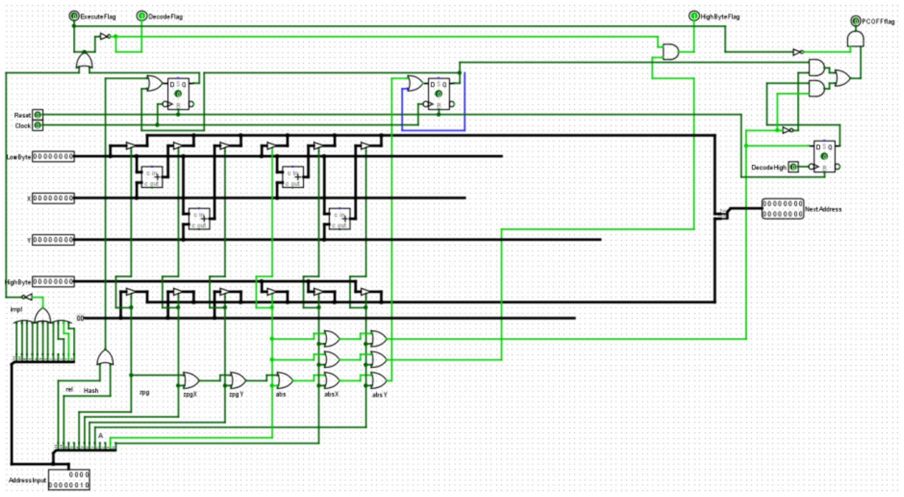


Fig. 30. Absolute Memory Required.

Indexed Memory Required. Finally Indexed Memory requires an additional Clock Cycle and therefore Additional D Type Flip Flop. Some additional wiring is required for Xind and indY Addressing Modes as they add the value of the X and Y register to either the first or second 16-bit number retrieved. The second D Type Flip Flop added for the Zeropage Addressing Modes can be used to distinguish what the current decode state requires. Resulting in the final design for the retriever circuit seen below.

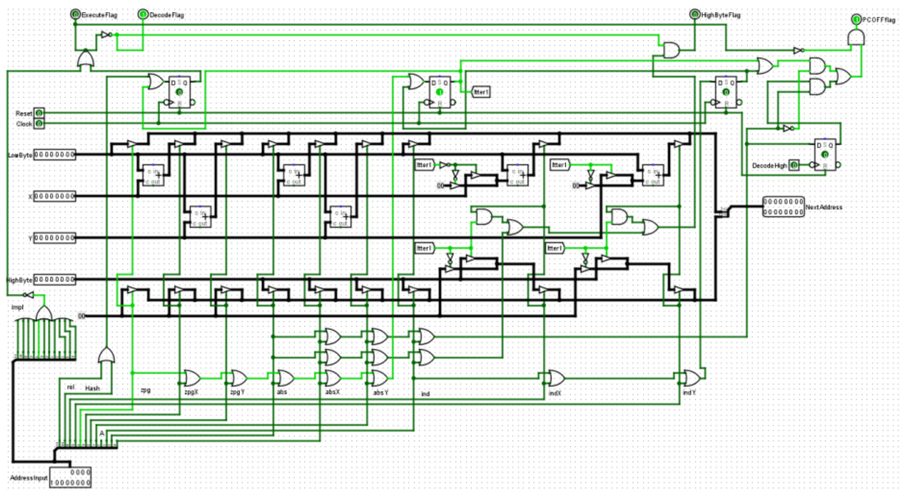


Fig. 31. Indexed Memory Required.

10.4 Updated Architecture

Before designing components for the execution state, the current architecture seen in Figure. 18. must be updated to contain these newly designed components.

Decoder Implementation. To add the decoder a wire is connected from the output of the CIR to the 8bit input pin of the decoder. The outputs of the decoder are merged into an address wire with each bit representing one of each of the addressing modes and three instruction wires which have individual bits representing each of the instructions.

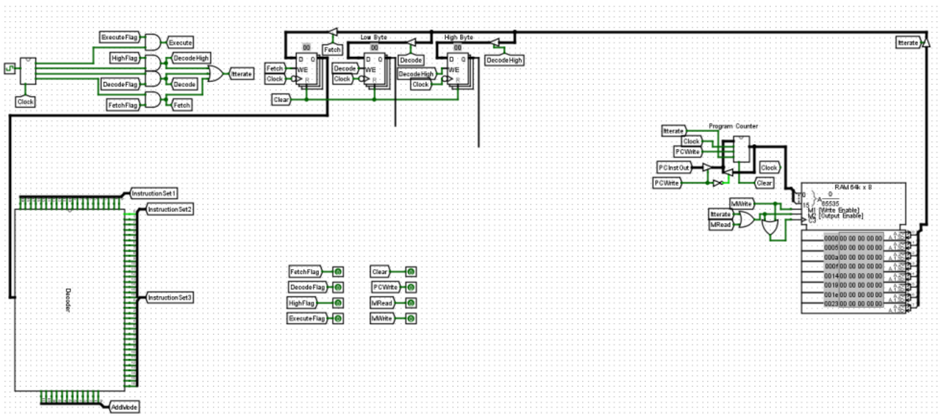


Fig. 32. Decoder Implementation.

Retrieve Implementation. Similarly to the previous step the retriever is added to the FPGA its appropriate inputs and outputs are connected.

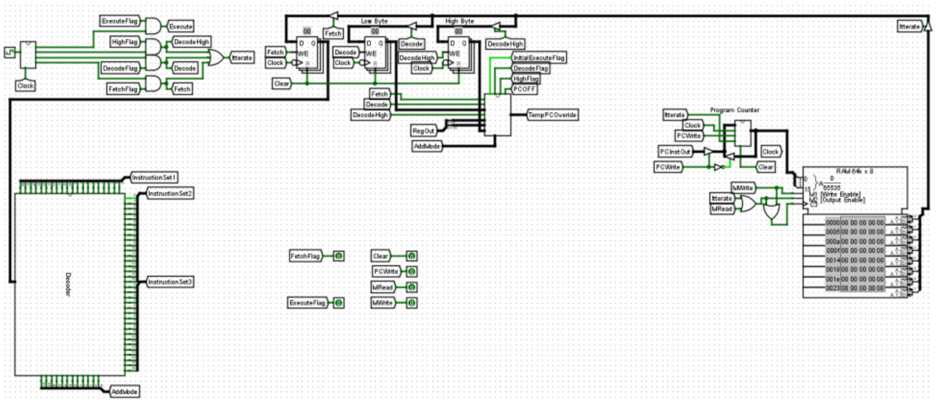


Fig. 33. Retriever Implementation.

Some additional components are still required for the retrieved component to work. Firstly, the component requires an input from the X and Y register. Which are implemented with the following circuit.

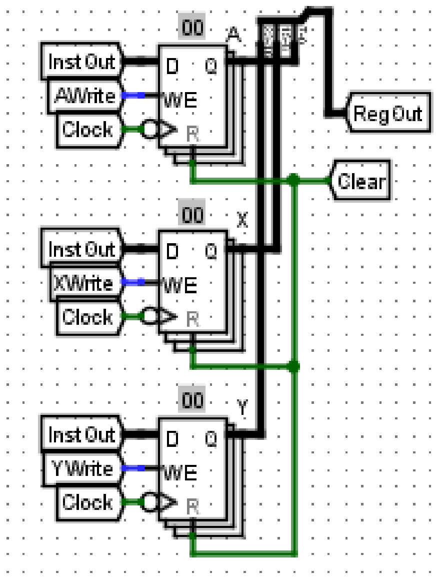


Fig. 34. Accumulator, X and Y Register Implementation.

Next some alterations must be made to the program counter. As previously explained during the building of the retriever certain addressing modes don't want to increment the program counter. Additionally addressing modes that retrieve from an indexed memory location will need to overwrite the program counter however after the decode state is complete if the program counter does not overwrite itself back to its original location these addressing modes will also perform a jump.

To accomplish this an additional alternate program counter will be added. Then if the retriever requests a memory location using the PCOFF wire the initial PC is turned off by preventing the inputs from reaching it and buffering its output to not cause data collision. Then the alternate Program Counter will turn on by doing the opposite and updating its location to the required memory location.

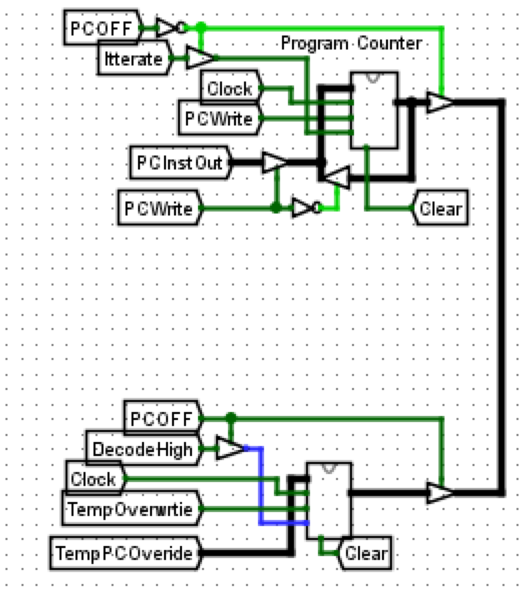


Fig. 35. Updated Program Counter.

The FPGA with implemented retrieve can be seen in Figure. 36.

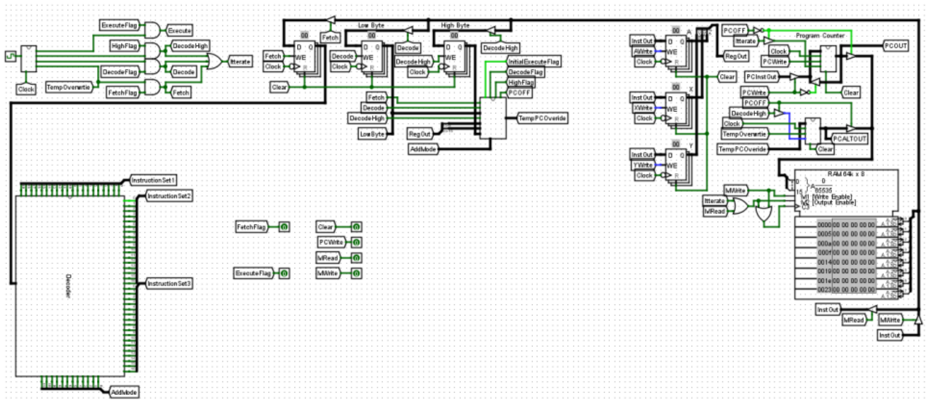
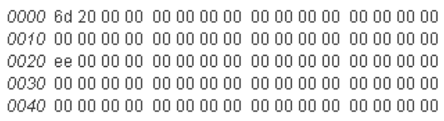


Fig. 36. Final Retriever Implementation.

To test the current FPGA the test program shown in Figure. 37. can be run. This is an instance of the test program described earlier in this document with 6d being the instruction ADC, abs and the next two memory locations 20 and 00 represent the memory location 0020 and the value at this location, ee being the value that should be added.



```
0000 6d 20 00 00  00 00 00 00  00 00 00 00  00 00 00 00
0010 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00
0020 ee 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00
0030 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00
0040 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00
```

Fig. 37. Retriever Test Program 1.

To verify the current iterations of the FPGA this program should result with the CIR containing the Instruction 6d and the MBR initially containing the address 0020 until the lowbyte is updated with the execution value ee.

Figure. 38. Shows the first cycle the CIR contains 6d and MBR contains 0020 with the PCOFF wire activated for the next decode state.

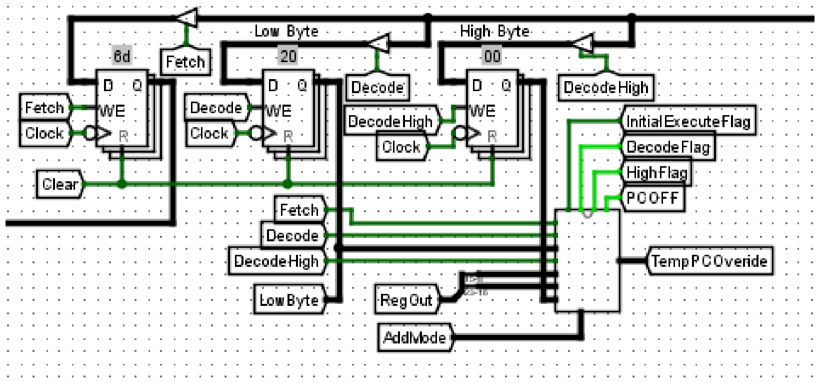


Fig. 38. Retriever Test Program 2.

Figure. 29. Shows the second and final clock cycle The Final contents of the lowbyte is ee which is the value that this instruction would use for execution. Additionally, the retrieve has raised the initialExecuteFlag indicating the instruction is ready to be executed.

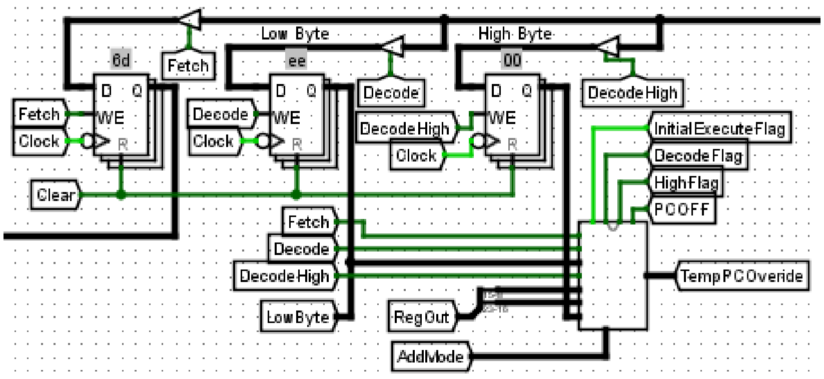


Fig. 39. Retriever Test Program 3.

More Components Pre-Execution. Before the FPGA can execute instructions, the remaining registers must be implemented these are the Processor Status Register and the Stack Pointer.

First the stack Pointer can be easily implemented the same way as the X, Y and Accumulator registers were implemented shown in Figure. 40.

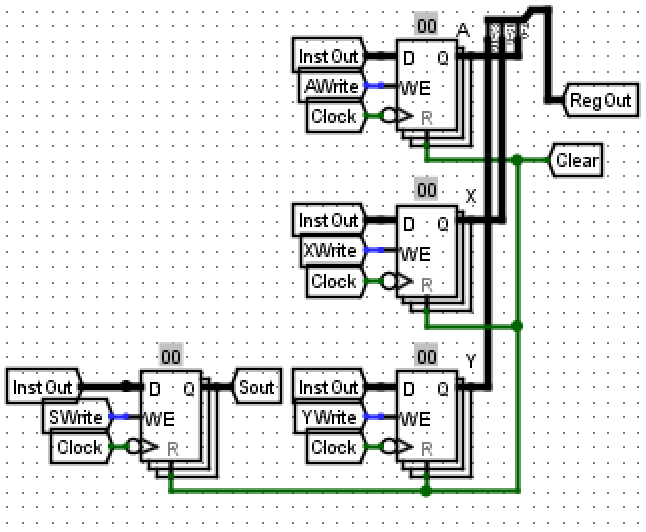


Fig. 40. Stack Pointer Implementation.

The Processor Status Register is harder to implement because it can be written to in multiple ways. First it can be written to like the previous register with an 8-bit value overwriting its contents. It is worth noting that the P register is a 7-bit register but when reading and writing to the entire register an 8th bit is padded onto the data value.

To accomplish this a D type Flip Flop is used to store each of the registers bits when writing from an 8-bit value the Register Input pin must be active this opens AND gates and allows for the data values and execute signals to reach their respective register.

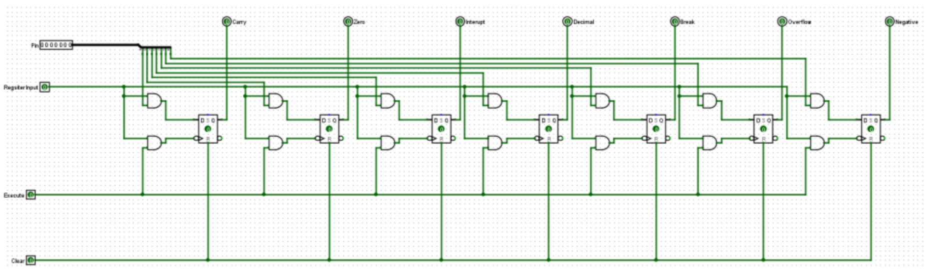


Fig. 41. Processor Status Register Implementation 1.

Next certain instructions allow for individual bits to be raised or cleared. The circuit shown in Figure. 42. is used to determine which input is being accessed. If RegisterInput is active the output will equal Input1 however if RegisterInput is not active Input2 the output will reflect input2.

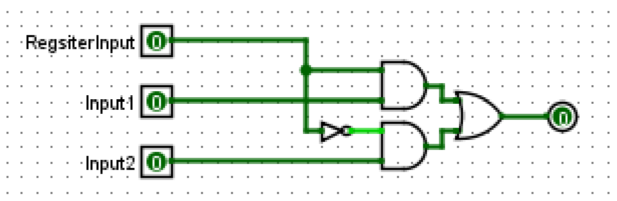


Fig. 42. Input Alternator.

Replicating this circuit before the data input, and clock input for each D type flip flop allows for these flags to be raised. Then NOT RegisterInput AND clear input allows for flags to be cleared.

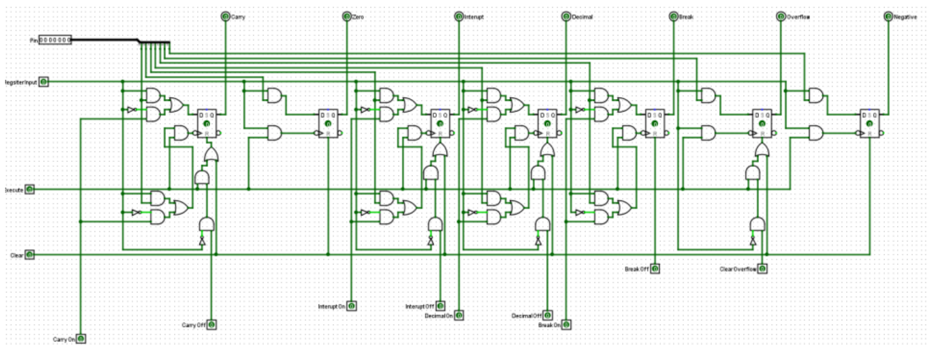


Fig. 43. Processor Status Register Implementation 2.

Finally, the last way of writing to the P register is by writing to individual data bits. Bits within the P register represent the current state of the FPGA. The zero flag is used to represent if certain instructions result in 00 being stored in a register or memory. For example, if the ADC instruction performs FF + 01 it will result in 00 being stored in the accumulator the P register will then raise the zero flag however if the ADC instruction performs FE + 01 it will result in FF being loaded into the accumulator because the output is not a zero the zero flag must be cleared.

This is achieved by having two inputs, the value and then read signal. Again, using the circuit from figure 43 to allow for the P register to alternate inputs to specific flags, the read signal can signal a flip flop to store the value input. This concludes the P register design shown in Figure. 44.

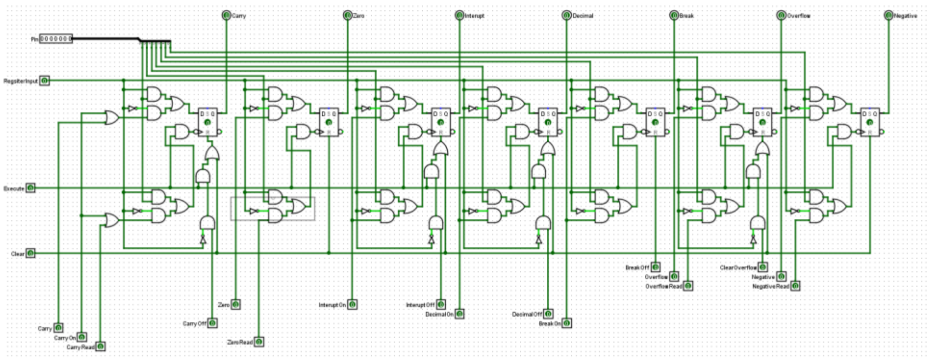


Fig. 44. Final Processor Status Register Implementation.

The current iteration of the FPGA can be seen in Figure. 45. It contains all the components required to begin designing the execute state.

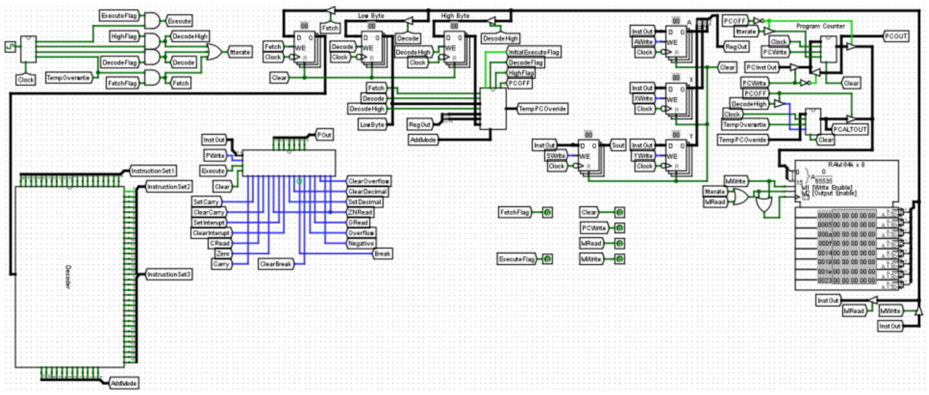


Fig. 45. Pre-Execution FPGA.

10.5 Execute State

Transfer Instructions. The first component to be designed will be the transfer component. Responsible for the following instructions: LDA, LDX, LDY, STA, STX, STY, TAX, TAY, TSX, TXA, TYA and TXS.

First all these instructions must be written from a register or memory to a register or memory so a mechanism to output the correct registers contents depending on instruction is designed. Using buffers the circuit in Figure. 46. can alternate between no outputs and the two registers depending on the instruction.

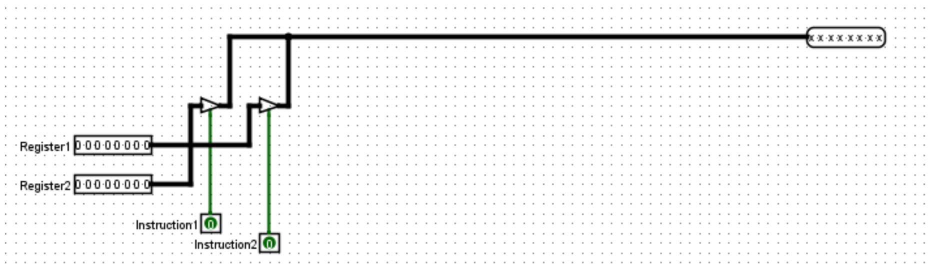


Fig. 46. Transfer Instructions 1.

Next the instruction must raise an output pin to indicate which register it is writing two in Figure. 46. we could assume instruction1 causes Register2 to update its contents to equal Register1. Therefore, it must now raise a flag to tell Regster2 to read from the output wire.

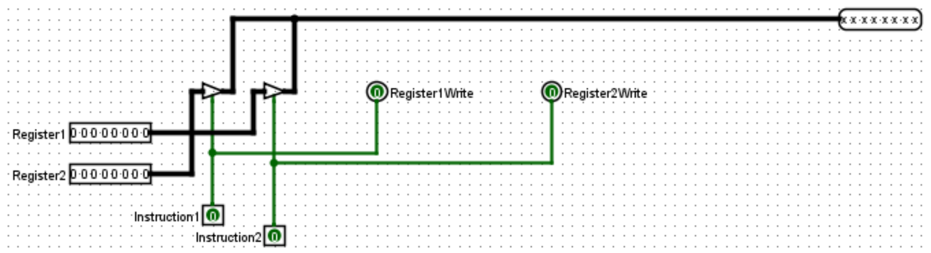


Fig. 47. Transfer Instructions 2.

Using the design in Figure. 47. and repeating the patterns we can produce a circuit capable of reading and writing to registers according to the 12 instructions that the transfer instruction component can execute.

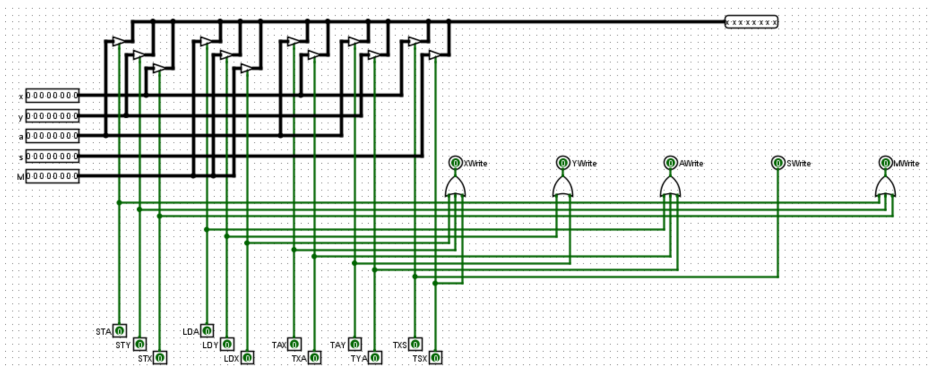


Fig. 48. Transfer Instructions 3.

The component is not yet complete. Three additions must be added to successfully execute these instructions. First While reading from memory for instructions like LDX, which writes from memory to the x Register, is relatively simple because the previous decode state loads the required memory value into the MBR lowbyte. Writing to Memory is potentially more difficult. The desired memory location could be stored in two locations either the addressing mode was an immediate memory location required like LDX, # in which case the memory location is stored in the program counter, or any other addressing mode was used in which case the memory location is stored in the alternate program counter. To solve this whenever writing to memory an additional output pin must be raised to keep the PC off if it is already off.

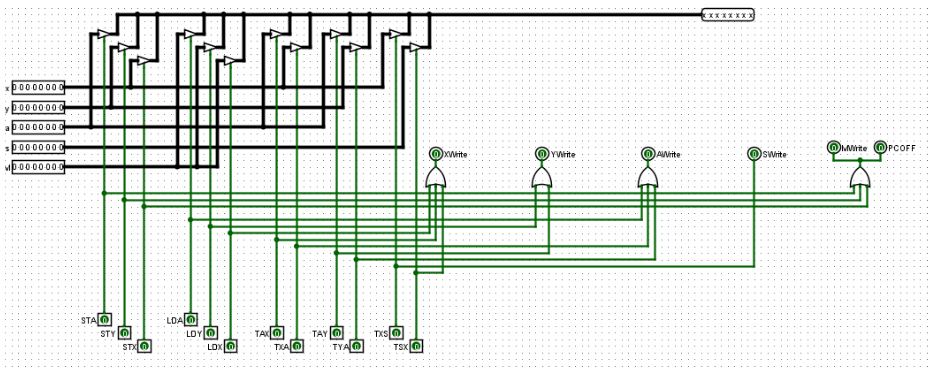


Fig. 49. Transfer Instructions 4.

Next because instructions can take different amounts of time to execute the circuit needs to indicate when it has finished executing so the FPGA can raise the FetchFlag for the next instruction.

All transfer instructions take exactly one executive state to complete so can immediately indicate they have finished executing.

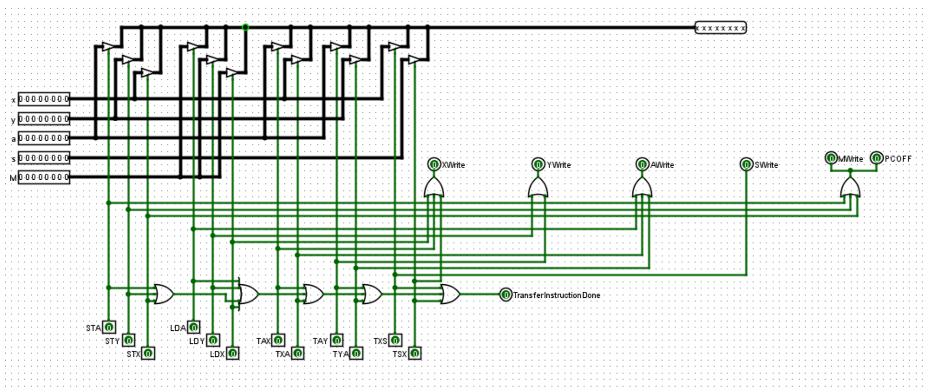


Fig. 50. Transfer Instructions 5.

Finally, most 6502 instructions will raise and lower flags withing the Processor Status Register for example all transfer instructions that write to registers will set the zero and negative flag to 1 if the value being transferred is a negative or a zero and to 0 if not.

The method for raising and lowering flags will be determined externally but will re-quire an indicator when a transfer instruction requires flags to be set.

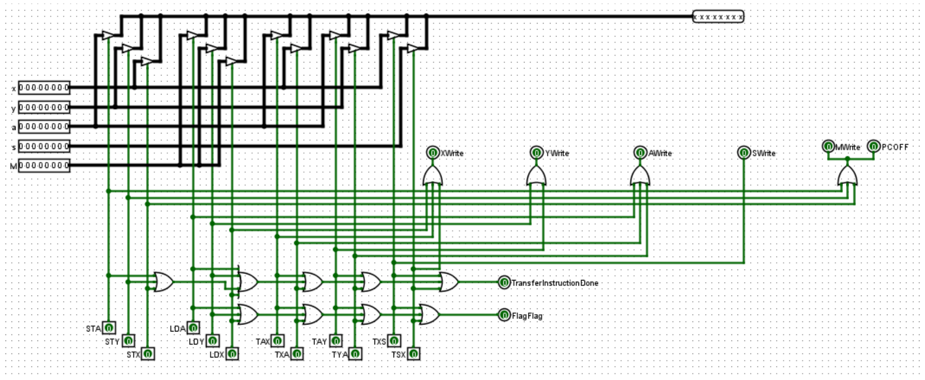


Fig. 51. Final Transfer Instructions.

Increment Instruction Component. The Increment Instruction Component has a very similar design to the transfer instruction component however instead this time the instruction reads and writes to and from the same register after having a constant of one either added or subtracted from the input value.

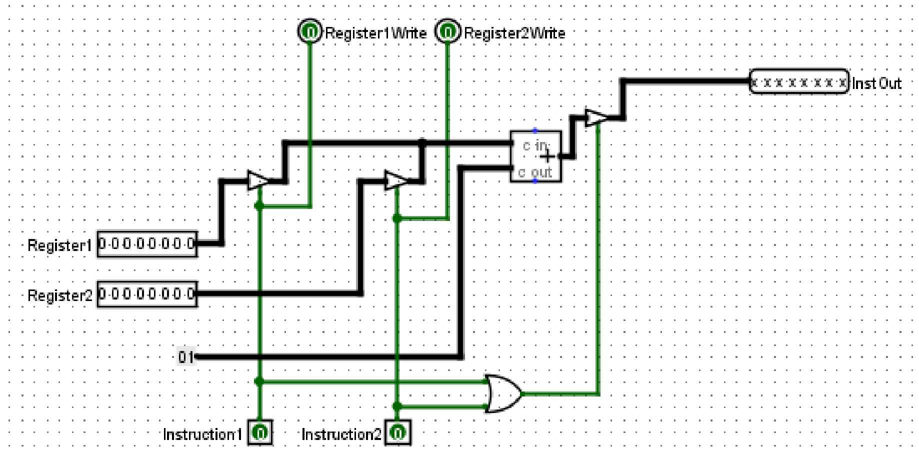


Fig. 52. Increment Instructions 1.

When designing these components, it is important that the output remains empty unless an instruction input pin is active. Later the output from each of these components will be merged into an instruction bus and components that leak data into the bus will cause data collisions. To mitigate the adder leaking data onto the data bus an additional buffer must be added to after the adder.

Again, the Increment Instructions only require one execution state to complete so they instantly indicate the instruction has finished and the fetch flag should be raised. However, all Increment Instructions affect the negative and zero flag because of this the InstDone indicator can also be used to indicate to set these flags appropriately. The final design for the increment instruction component can be seen in Figure. 53.

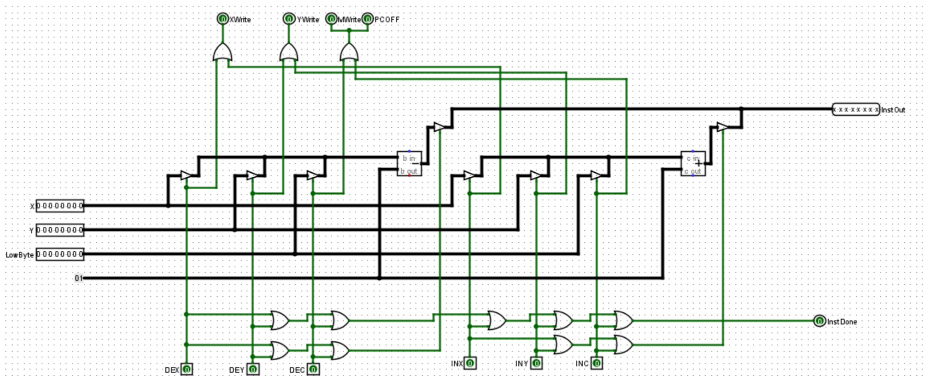


Fig. 53. Final Increment Instructions.

Logical Instruction Component. The logical instructions are incredibly simple including AND, ORA and EOR that pass the indicated memory value stored in the low byte and the contents of the Accumulator through a AND, OR or XOR mask respectively. Storing the output back in the accumulator. Using the same method as the prior components produces this circuit.

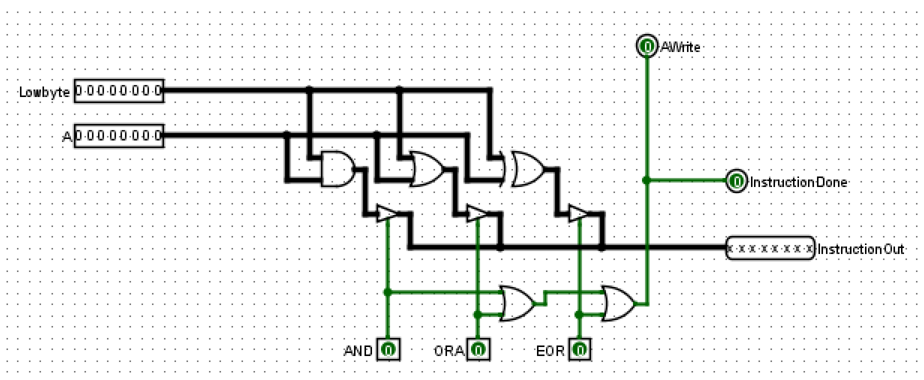


Fig. 54. Final Logical Instructions.

The logical instructions all require exactly one execution step so the instructiondone pin can be immediately activated. The instructions also only affect the negative and zero flag allowing the instruction done pin to also indicate these flags to be affected.

Rotation Instruction Component. The rotation instructions include ASL, ROL, LSR and ROR which require all the bits in an 8-bit number to be shifted one place to the left or right and stored in the accumulator or memory. The in Figure. 55. circuit can be used to shift all bits in a number to the left. And will be used to complete these instructions.

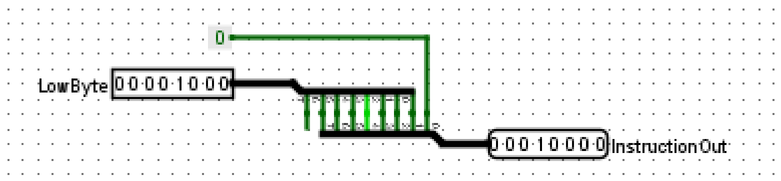


Fig. 55. Rotation Instruction 1.

Currently this circuit sets the rightmost bit equal to 0 however the ROL instruction requires this bit to be set equal to the state of the carry flag. To accomplish this an input alternator as shown in Figure 42. Is required. The circuit in Figure. 56. sets the right most bit equal to 0 or the carry flag depending on the instruction input.

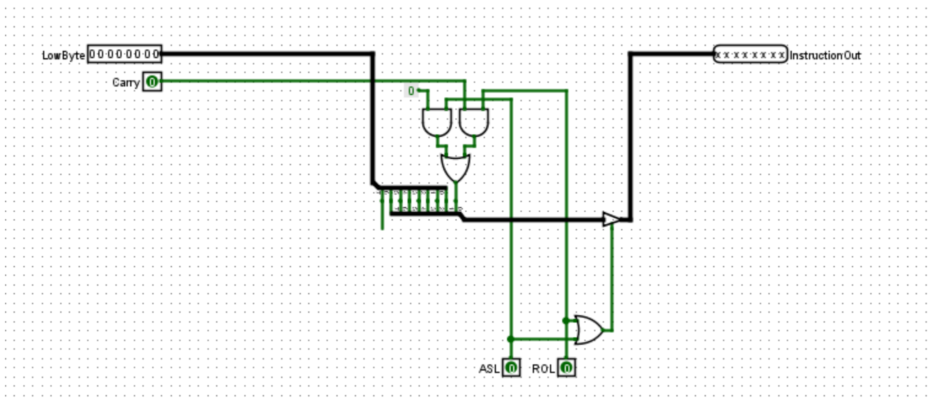


Fig. 56. Rotation Instruction 2.

Now the circuit can be expanded to include the right shifting instructions. Additionally, this circuit must immediately output to indicate the instruction has finished as rotation instructions take exactly one execute state to complete. Finally, when shifting left the 8th most bit is currently lost but some Rotation Instructions store this lost bit in the carry flag so a wire must be added to extract the extra bit to be given to the carry flag later.

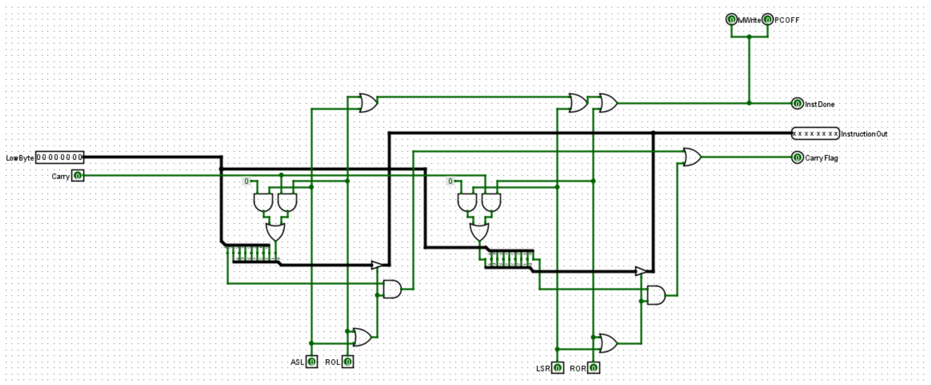


Fig. 57. Rotation Instruction 3.

This circuit works for most rotation instructions, however there is a fringe instance where a rotation instruction can be applied to the accumulator using the A addressing mode. The circuit in Figure. 58. is used to alternate between the low byte and the accumulator using an input pin AccumulatorTrigger which will be activated during the fetch state if the A addressing mode is decoded.

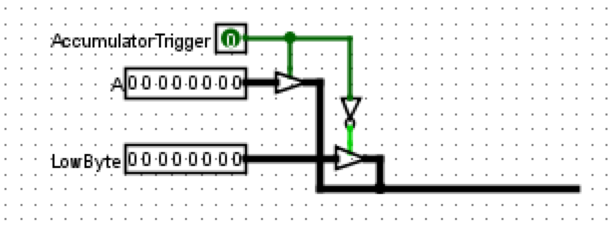


Fig. 58. 8-bit input alternator.

Finally, after adding AND gates to alternate between writing to Memory of the Accumulator the Rotation Instruction circuit is complete. Final Rotation Instruction Component shown in Figure. 59.

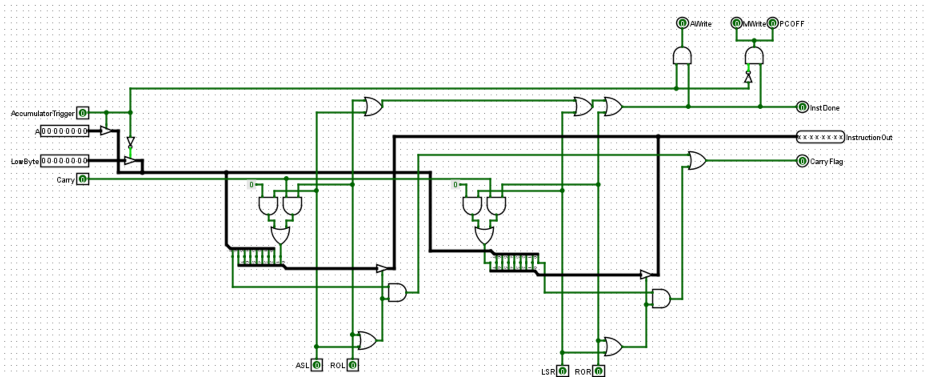


Fig. 59. Final Rotation Instruction.

Arithmetic Instruction Component. Arithmetic Instructions include ADC, SBC, CMP, CPX and CPY. The first circuit seen in Figure. 60. outputs either the accumulator plus the lowbyte and the carry bit or the accumulator subtract the lowbyte and the carry bit. This will be able to execute the ADC and SBC instructions. The relevant output pins have been added for these instructions as well.

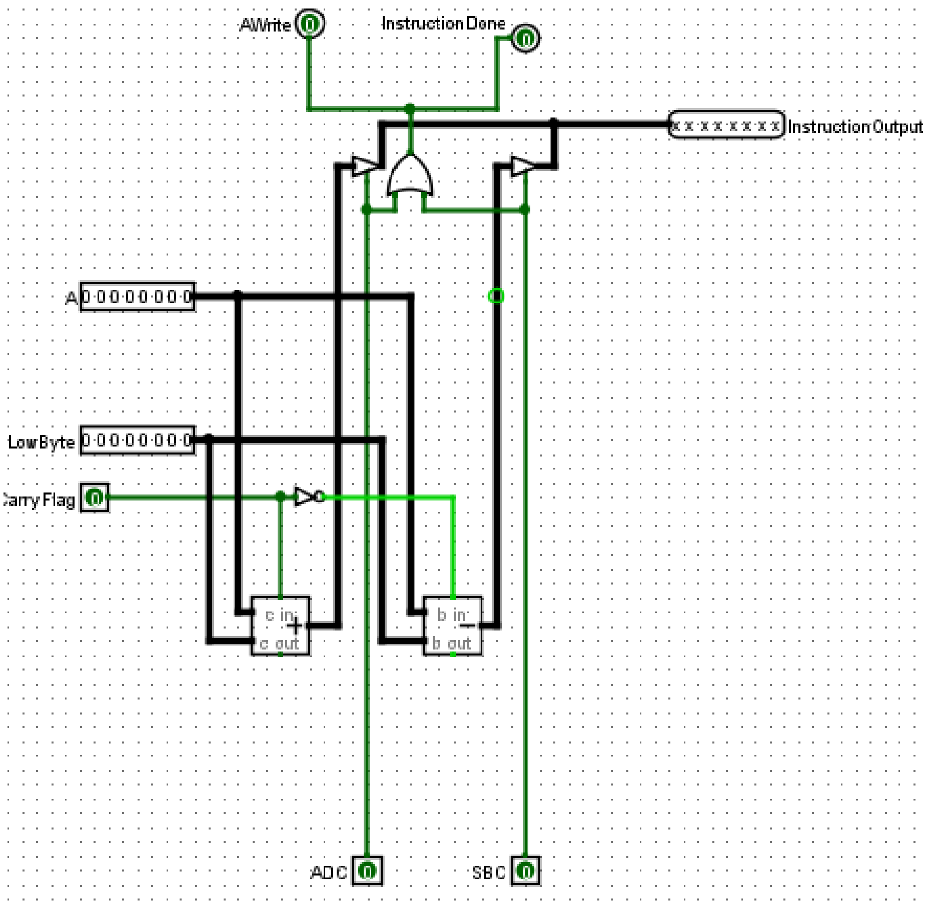


Fig. 60. Arithmetic Instruction 1.

Next the CMP, CPX and CPY instruction finds the difference between a memory location and the value stored in either the A, X or Y register. However, these instructions do not write to any registers or memory only updating the relevant flags in the P register.

To accomplish this buffer is added before the 8-bit subtractor that can be used to alternate between the three registers depending on instruction input. Extra wires are also added to indicate if the overflow or carry flag should be raised.

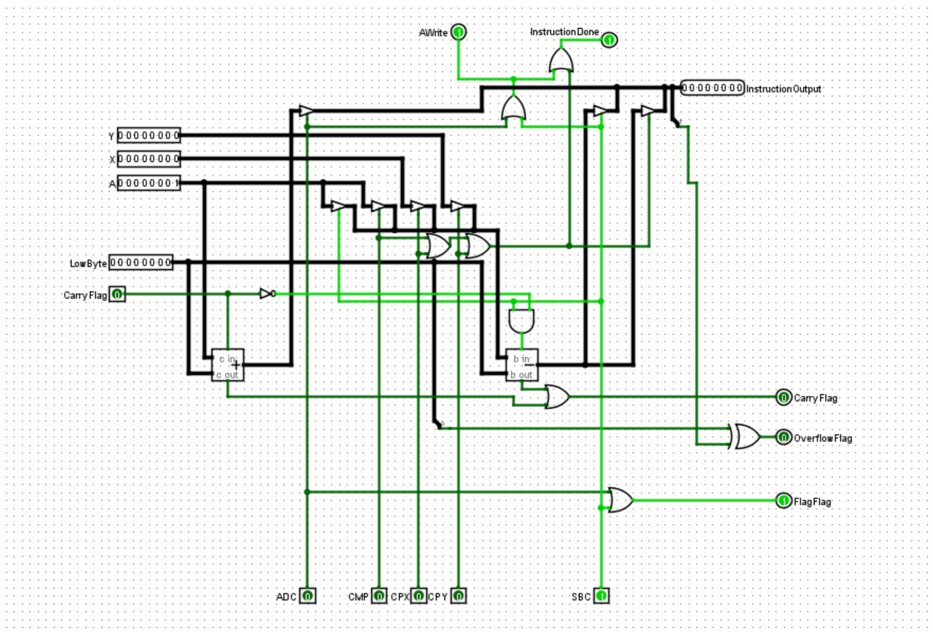
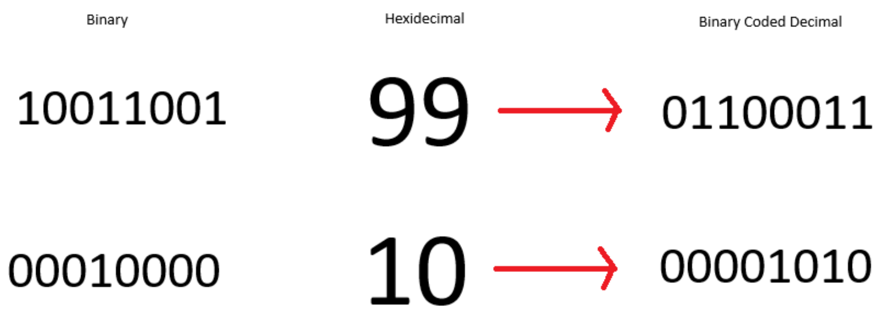


Fig. 61. Arithmetic Instruction 2.

This circuit can execute the arithmetic instructions in most situations. However, one of the flags within the P register is the decimal flag which when active converts all arithmetic instructions into using binary coded decimals.

What are binary coded decimals? Binary coded decimals are a method of interpreting 8-bit numbers as decimal values instead of binary ones. For example, the value 10011001 or 99 In hexadecimal has a value of 153. However, when interpreting as a binary encoded decimal it will have a value of 99. Effectively the hexadecimal number is interpreted as a decimal number.



Binary	Hexidecimal	Binary Coded Decimal
10011001	99 ⟶	01100011
00010000	10 ⟶	00001010

Fig. 62. Example of binary coded decimals.

First an explanation on how to add binary coded decimals. This example will add the numbers 35 (00110101) and 45 (01000101). To begin the two 8-bit numbers are split into two pairs of 4-bit numbers, The leftmost 4-bit numbers and rightmost 4-bit numbers can be added together.

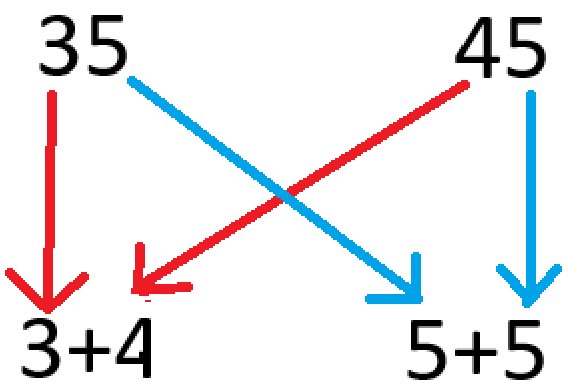


Fig. 63. Binary Coded Decimals Addition 1.

In our example this results in 3(0011) + 4(0100) = 7(0111) and 5(0101) + 5(0101) = a(1010). Next, we must determine if there was any overflow. The best way to do this is to subtract a(1010) and see if the calculation results in a NOT overflow because if the values being added together total to greater than a digit must be carried.

continuing the right digit a(1010) subtract a(1010) equals 0(0000) causing no overflow this means an additional bit must be carried to the left digit and the new value 0(0000) should be used for the rightmost 4 digits of the output. Next the left digit plus carry 8(1000) subtract a(1010) equals d(1101) which overflows meaning the original calculation of 8 was correct.

Finally, the two 4bit halves can be merged into an 8-bit number resulting in 80(10000000) which when interpreted as a hexadecimal is completely wrong however is correct when interpreting as a binary coded decimal means 35+45 = 88.

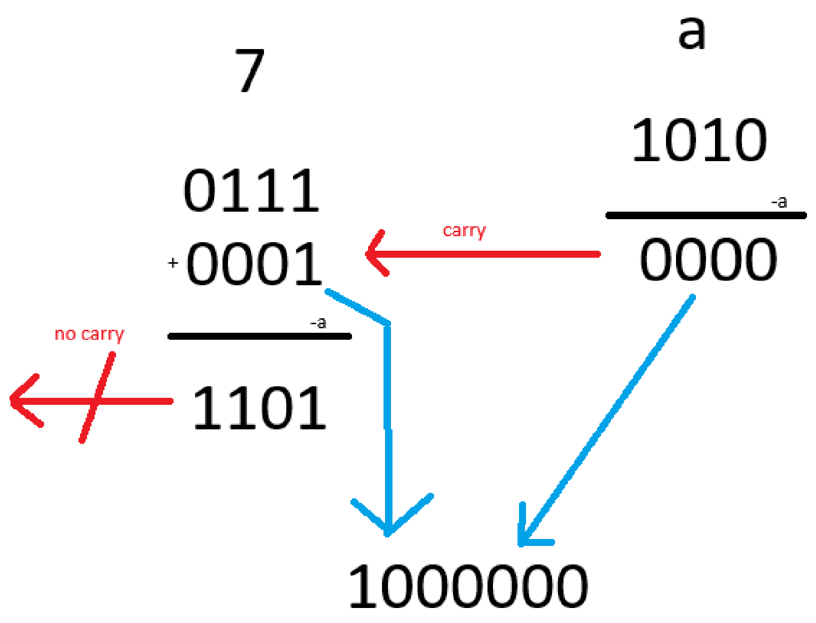


Fig. 64. Binary Coded Decimals Addition 2.

Next subtracting binary coded decimals for this we will use the example 30(00110000) subtract 19(00011001) the process starts the same way with the left and right bits of each number being subtracted from each other. This results in 2(0010) and 7(0111) with the right digit carrying causing an extra 1(0001) to be subtracted from the right digit.

2
7
0010
0111
- 0001

Fig. 65. Binary Coded Decimals Subtracting 1.

First, we will resolve the right digit because it previously overflowed, we instead subtract 6 from this value and if that causes an overflow, we can tell the original value didn't actually overflow and the original value can be output. In this case 7(0111) subtract 6(0110) equals 1(0001) with no overflow so this digit will represent the rightmost 4bits of the later 8-bit number.

Next the left digit did not overflow so instead we must subtract a(1010) and, if that causes a NOT overflow it did overflow and the new value must be used in this case 2(0010) subtract 1(0001) subtract a(1010) equals f(1111) causing an overflow. This means the original value of 2(0010) subtract 1(0001) should be used for the left 4-bits.

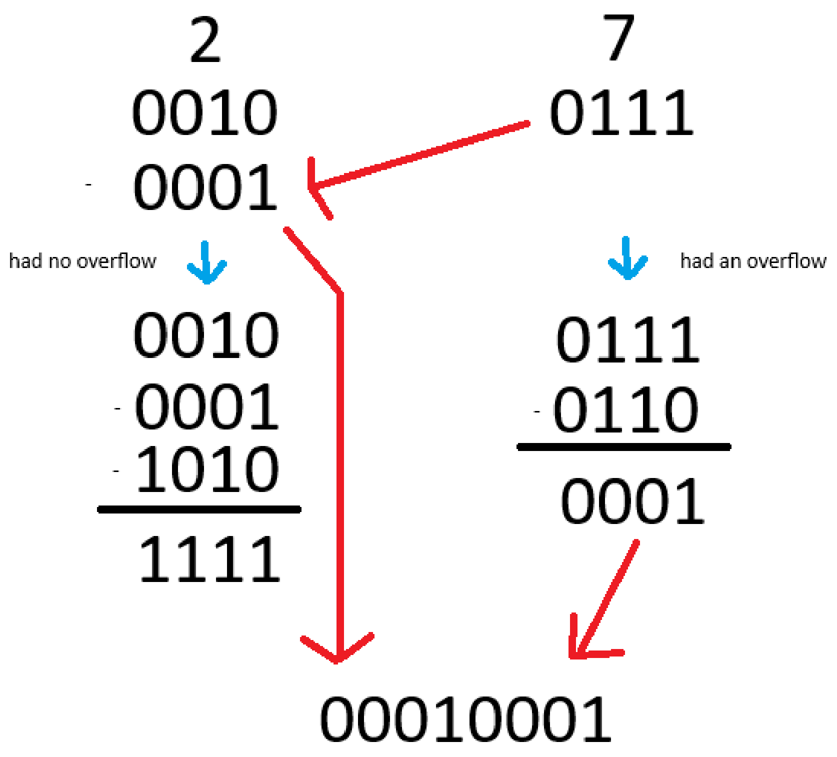


Fig. 66. Binary Coded Decimals Subtracting 2.

The resulting in 30 subtract 19 equaling 11(00010001), completely wrong if interpreted as a hexadecimal but correct if interpreted as a binary coded decimal.

To implement this as a circuit an input pin is added to relay if the decimal flag is raised. This attaches to an 8-bit input alternator that determines if the regular addition and subtraction circuit are connected to the output pin or if the alternative binary decimal mode addition and subtraction circuit connect to the output pin. The resultant and final circuit viewable in Figure. 67.

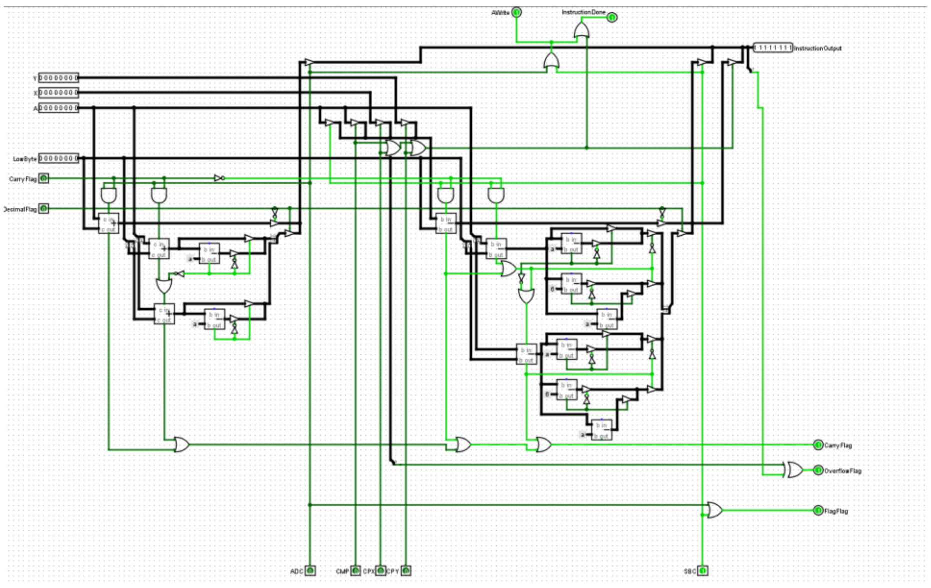


Fig. 67. Final Addition Instruction.

Branching Instruction Component. The branching instructions are comparatively simple to implement. The circuit in Figure. 68. only allows data to pass if the carry flag is NOT set.

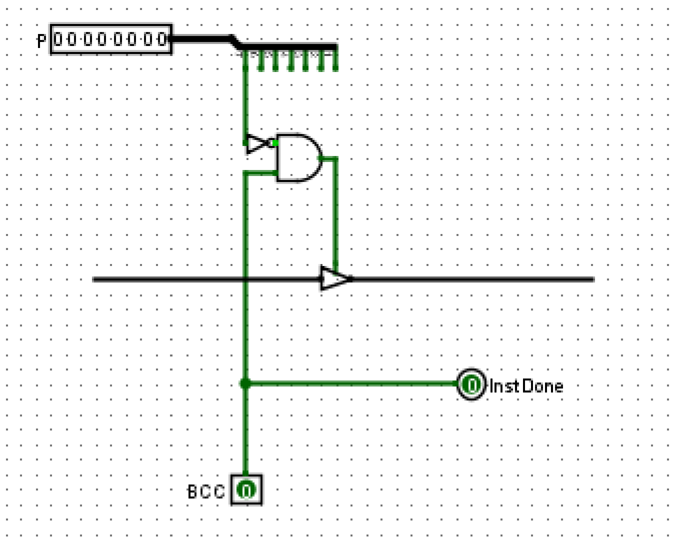


Fig. 68. Branching Instruction 1.

All branching instructions check if a specific flag is set or NOT set and then branch to a new memory location by adding the indicated value stored in the lowbyte to the contents of the program counter writing this back into the program counter. Unlike previous instructions this requires a 16-bit output pin as the Program Counter is a 16-bit register.

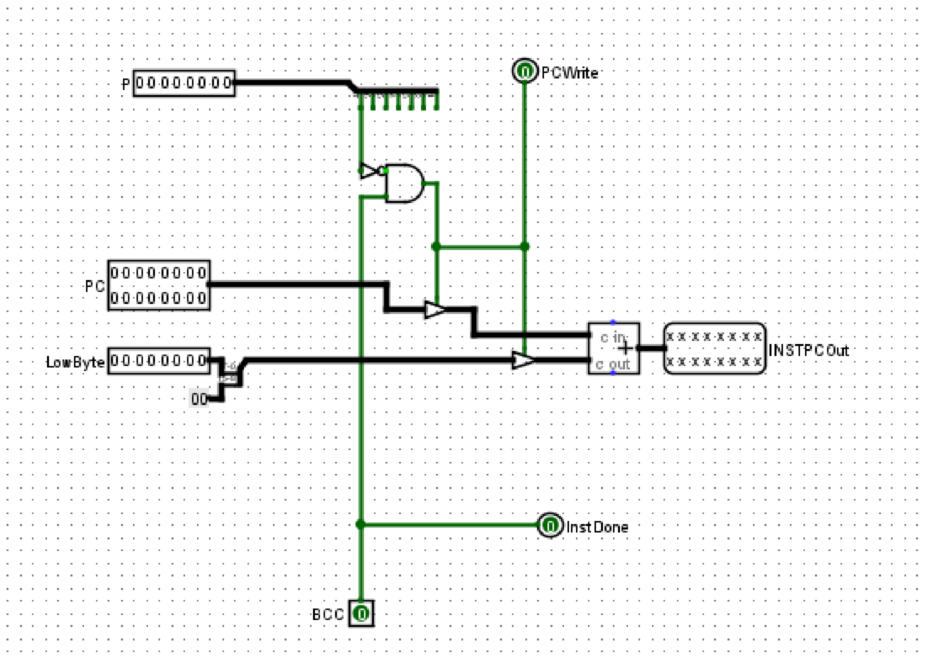


Fig. 69. Branching Instruction 2.

This design can then be repeated to execute the remaining Branch Instructions. The final Branch Instruction Component can be seen in Figure. 70.

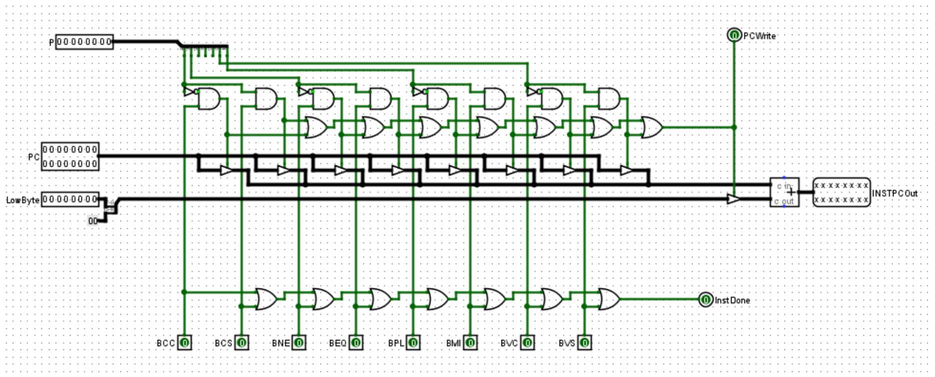


Fig. 70. Final Branching Instruction.

Stack Instruction Component The stack instructions include PHA, PHP, PLA and PLP that require either the P or A register to be read or written from memory. These instructions require multiple execution states to complete. The PHA and PHP instructions write to stack by increment the stack pointer, load the memory location indicated by the stack pointer, write the required value into stack. The PLA and PLP instructions do the opposite to read from stack.

Starting with the Input Counter shown in Figure. 25., we can begin to design this component starting with the first state which either decreases the stack point or load a memory location.

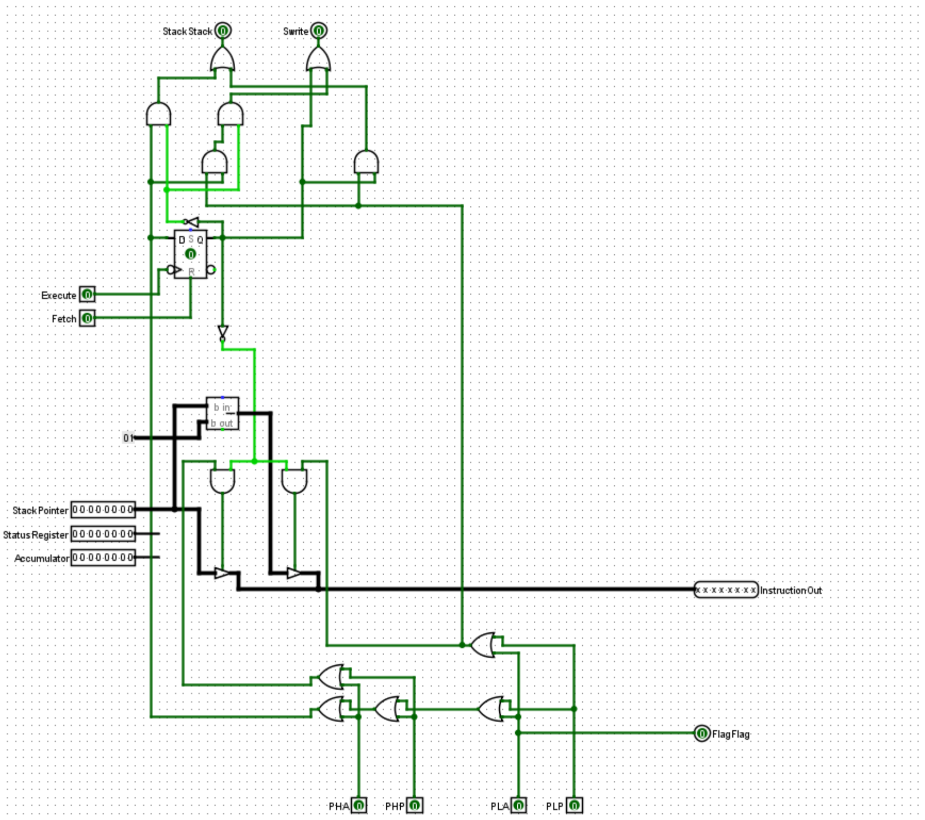


Fig. 71. Stack Instructions 1.

The output pin stackstack will be used to indicate that a new memory location should be loaded from the stack and like the decode state the fetch pin is used to reset the decoders during a fetch state as a new instruction will have begun.

The next step is like the first except the instructions written to stack are now incrementing the stack pointer and vice versa.

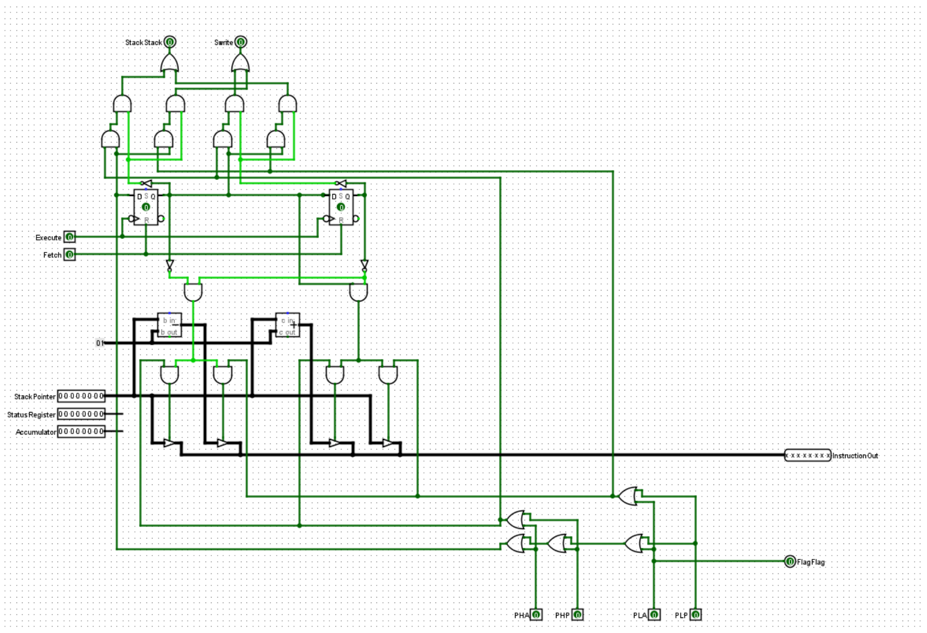


Fig. 72. Stack Instructions 2.

Finally for the last state all instructions will be outputting a memory location or a register content and writing to the opposite. It is important that when reading from memory no data is output onto the instruction bus to avoid data collision.

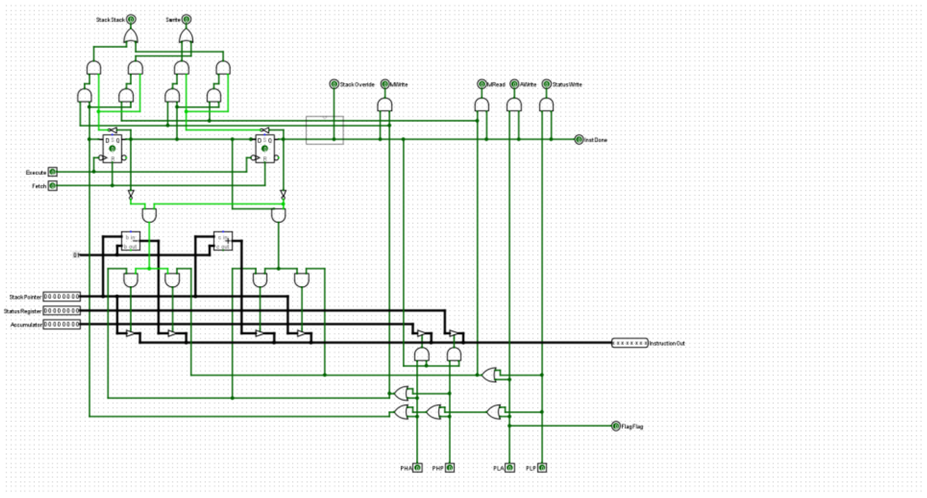


Fig. 73. Final Stack Instructions.

Jump Instruction Component. Jump instruction includes the JMP, JSR and RTS instructions. When jumping the lowbyte contains an arbitrary value that is not needed, The alternate PC contains the location the instruction needs to jump to. To complete a jump instruction all that must happen is the Program counter must be overwritten with the content of the alternate program counter. The circuit in Figure. 74. does exactly that.

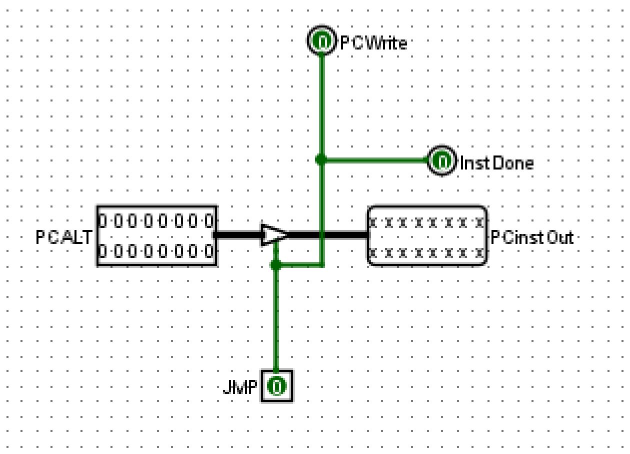


Fig. 74. Jump Instructions 1.

The circuit in Figure. 74. is able to complete the JMP instruction. The JSR instruction works the same however first pushes content of the program counter before jumping onto the stack. Using the input counter from Figure 25. In the same way done for the stack instructions the following circuit loads the content of the S register into memory, increments the S register, writes the first 8-bits of the program counter into memory. This is then repeated for the next set of 8-bits in the program counter, then the contents of the alternate PC can be overwritten into the Program Counter.

An additional register must be added to store the contents of the alternate PC. At the beginning of the first execution step of the JSR instruction the value is stored in a 16-bit jump register. This is necessary as during the process of writing to the stack the contents of the Alternate Program Counter may be updated.

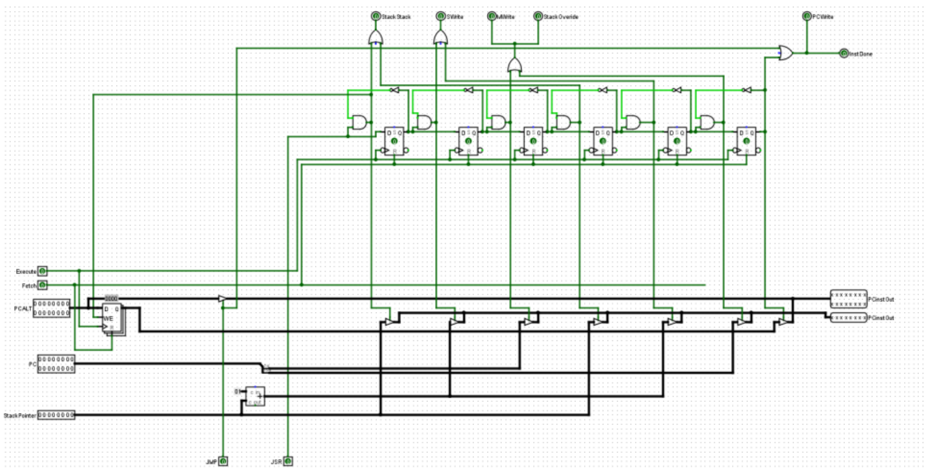


Fig. 75. Jump Instructions 2.

Finally, the RTS instruction does the opposite of the JSR instruction. The idea of these instructions is a programmer can jump to a memory location execute a subroutine and then return to the original memory location. This pair of instructions allow the programmer to jump to the same subroutine from multiple lines in their code and return to the different locations.

Like the stack instructions when reading from the stack the S register is decrement-ed, loaded into memory, and then read from. However, because the program counter is a 16-bit register and the value that must be returned to is split into two 8-bit values, when reading from memory these values must first be stored in temporary return registers which can then be merged into a 16-bit value when writing to the program counter.

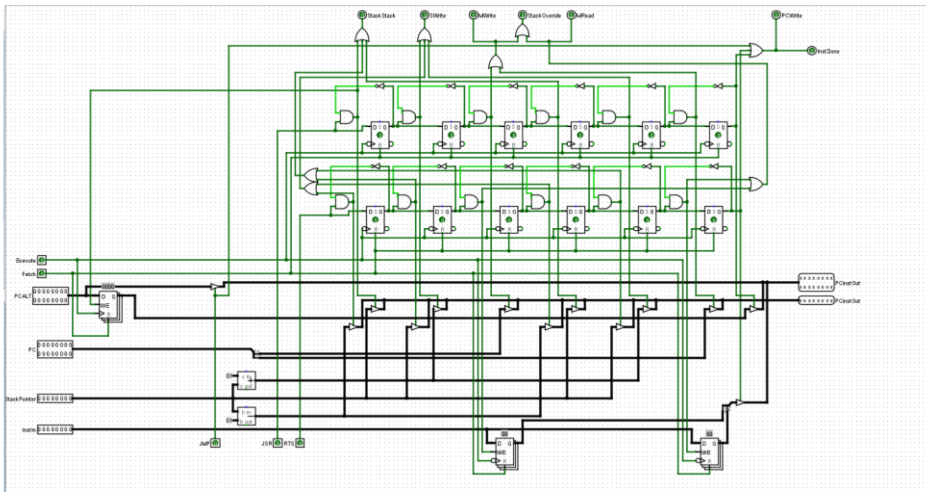


Fig. 76. Final Jump Instructions.

Interrupt Instruction Component. The interrupt Instruction includes BRK Interrupt causes an interrupt and RTI which returns from an interrupt. Interrupts are caused and returned from by raising and lowering the break flag. Before that the BRK instruction must push the contents of the program counter and the contents of the P register onto the stack. With the RTI instruction doing the opposite.

So again, to build this circuit the input counter shown in Figure. 25. will be used correctly time the order of execution and the same execution order when writing to the stack will be used. The final instruction that will be executed will be raising the break flag.

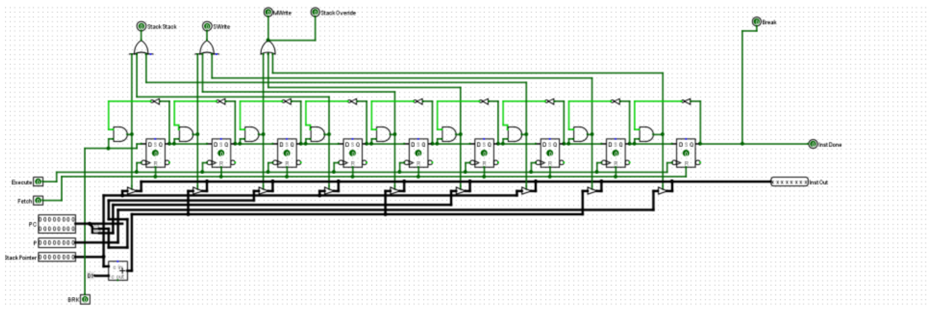


Fig. 77. Interrupt Instruction 1.

And using the same method as when returning from subroutines the RTI instruction can also be implemented.

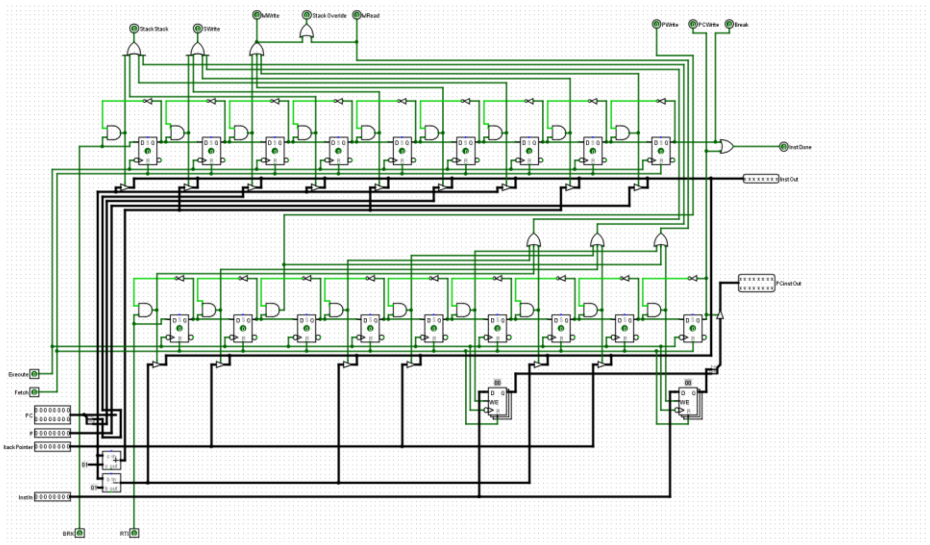


Fig. 78. Final Interupt Instruction.

Execute Component. The execute component can now be assembled connecting all components described earlier in section 10.5 to the necessary input and outputs, Resulting in the following component.

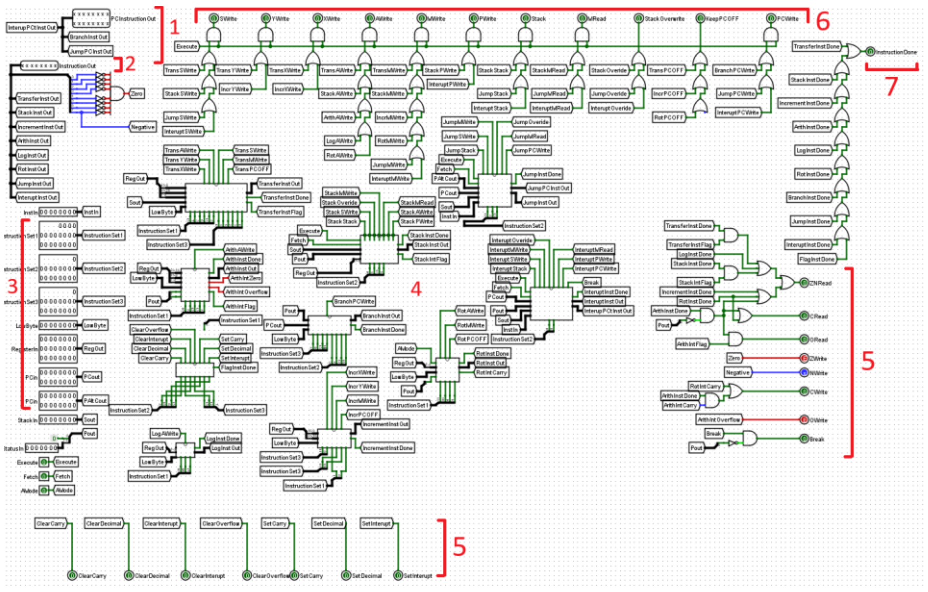


Fig. 79. Execute Component.

With numbers added to represent.

1. The 16-bit PC instruction bus is used to overwrite the address stored in the Program Counter.
2. The 8-bit instruction bus is used to convey outputs from the execute instruction to whichever register/ memory location is being written to.
3. All input pins are required for the components within the execute component.
4. The sub-components within the execute components described earlier in section 7 with their appropriate input and output pins connected to the wider architecture.
5. Flag operation and commands used to set, lower, or alter specific flags within the P register.
6. Output pins used to signal commands to circuits outside the execute components such as making registers writable to.
7. The InstructionDone pin used by the FPGA to indicate when an instruction finishes and to fetch the next instruction.

10.6 Final Architecture.

To finalize this FPGA four additional features are required: A circuit capable of re activating the fetch flag after the execute instruction finishes, A connection from memory to the execute component allowing the execute component to read and write to memory while executing, a connection allowing a memory location to be loaded from the stack pointer and an interrupt controller that allows interrupts to be manually triggered and interrupts to be returned from.

Fetch Flag Circuit. The circuit shown in Figure. 80. will turn the Execute Flag off after the end of the execute state. Using a D type flip flop to record clock inputs this circuit can clear this flag after the execute state has finished this prevents flicker. Then, after another clock input, the Fetch flag is raised allowing for the next instruction to be retrieved and resetting the retrieve and execute components ready for another clock cycle. This circuit then resets the circuit after the fetch state has completed by timing three clock inputs from the previous execute state.

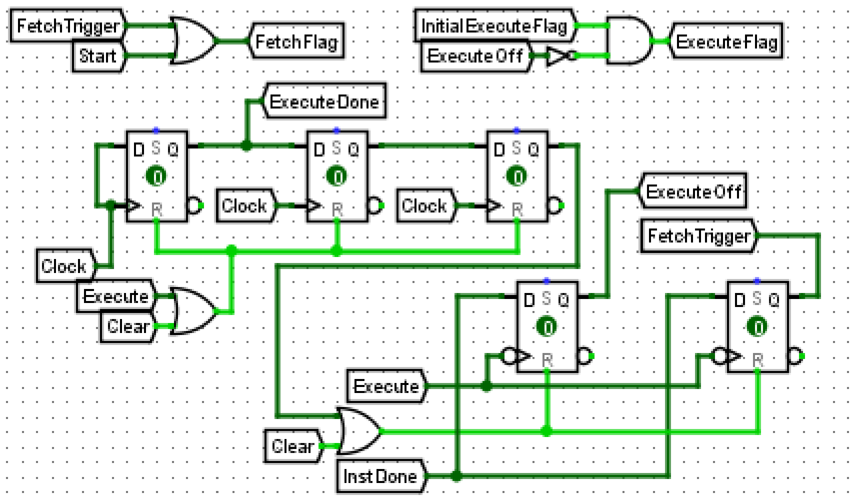


Fig. 80. Fetch Flag Raiser Circuit.

Expanded Memory Connection. This connection, displayed on the left of Figure. 81. allows the execution component to read and write to memory during the execute state. The buffers are important to prevent data collision during other clock state. The connection shown on the right of Figure. 81. can be used to store the contents of the stack pointer into the stack buffer register the during a stackoveride memory will load the memory location indicated by the stack buffer register.

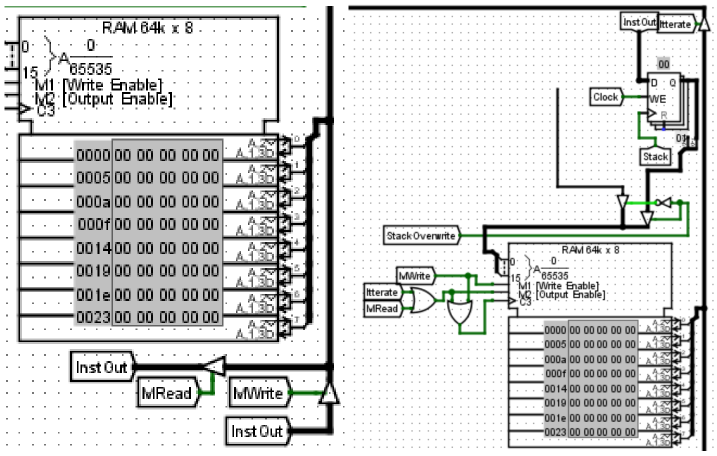


Fig. 81. Execute and Stack to Memory Connection.

Execution Implementation. The execution component can be added to the main architecture by wiring the appropriate input and output wires to its input and output pins.

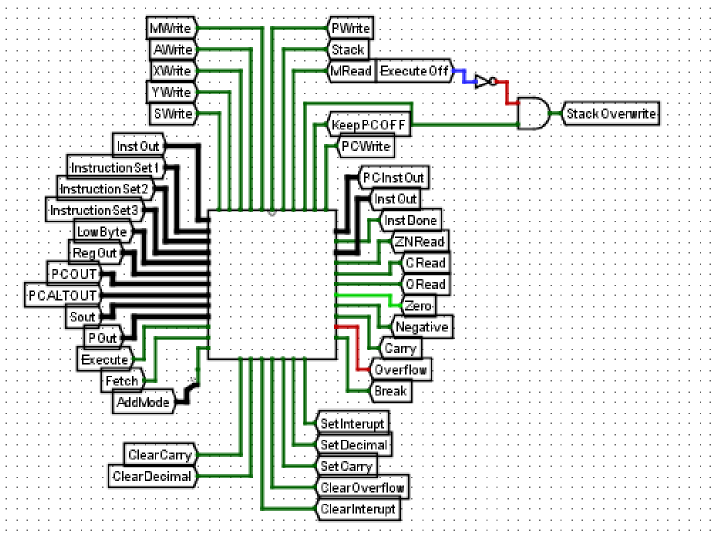


Fig. 82. Execution Component Implementation.

Interrupt Controller. The interrupt controller is split into three input pins. First the power pin causes the FPGA to be “on”. This is accomplished by preventing the clock signal from reaching the clock component by an AND gate. The AND gate allows the signal through as long as the break flag is off, AND the power pin is on. The power pin being turned off will also cause all registers to clear their contents. An output pin has been added to indicate when the FPGA is halted due to interruption.

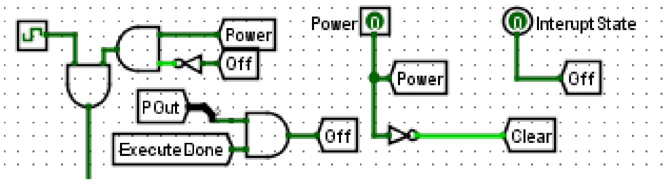


Fig. 83. Interrupt Controller Power Pin.

The manual interrupt pin functions by writing a one to a D type flip flop when activated. This removes the need to hold a continuous activation of the interrupt pin to cause an interrupt and removes the risk of a partial interrupt. The D-type flip flop then uses an 8-bit input alternate like the one shown in Figure. 62. To cause the next read instruction to be a 00(BRK) instead of a memory location. Once the instruction has been executed and the break flag raised the off signal cause the D-type flip flop to clear ready to receive another manual interrupt.

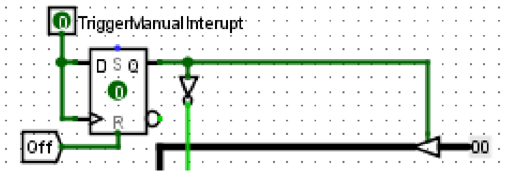


Fig. 84. Interrupt Controller Manual Interrupt Pin.

Finally, the return from interrupt pin using an AND gate will write a one to a D-type flip flop only if it is active AND the break flag is set. If the flip flop is successfully written to the break flag is cleared and the interrupt is override allowing the FPGA to execute a single instruction using the same input alternator, 40(RTI). Using an AND gate, its own input and an execute flag the flip flop can clear itself. This allows the Return From interrupt pin to reset itself ready for another activation.

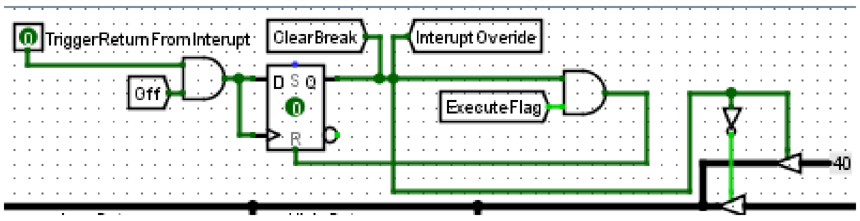


Fig. 85. Interrupt Controller Return from Interrupt Pin.

6502 Implementation.

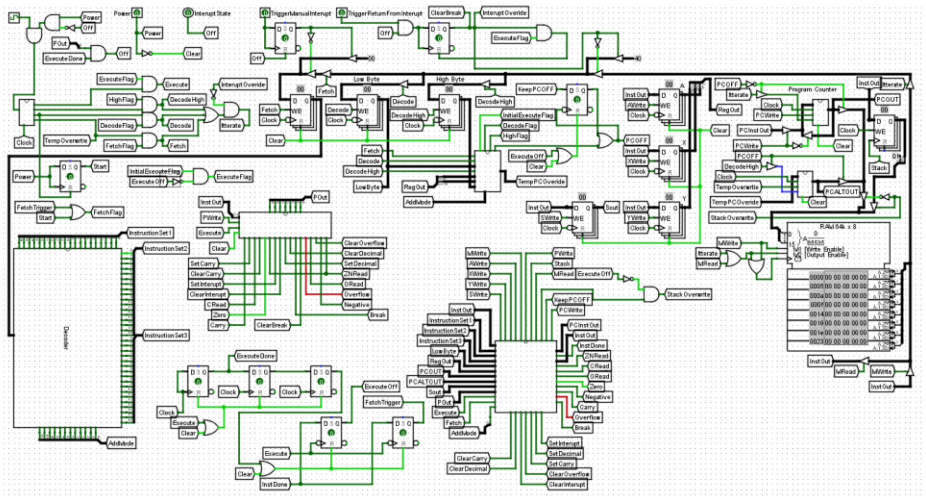
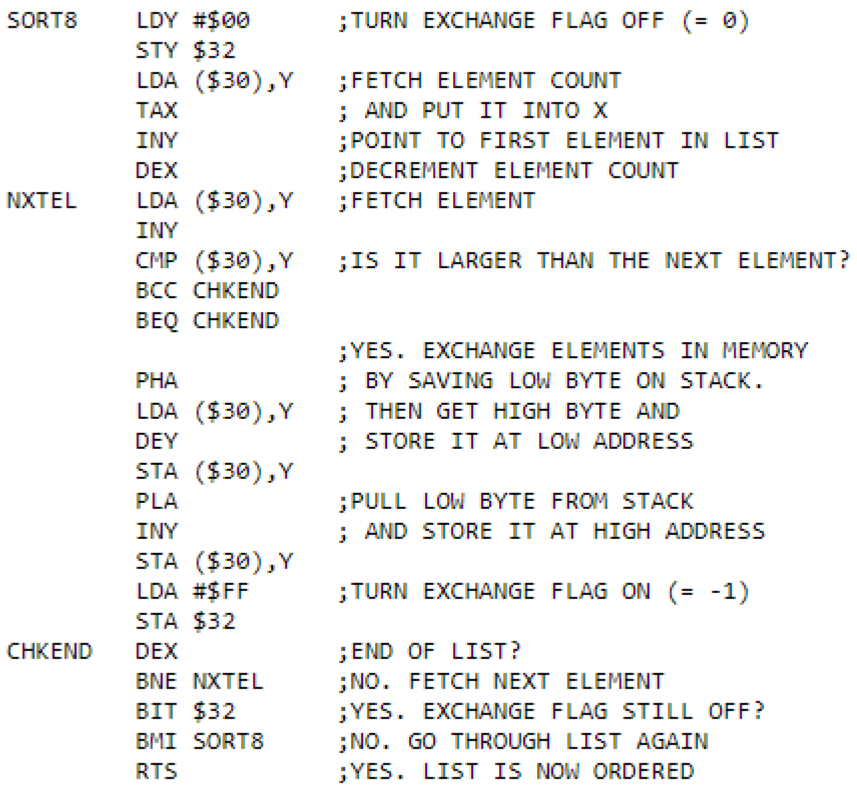


Fig. 86. Final 6502 Implementation.

11 Results and Discussion.

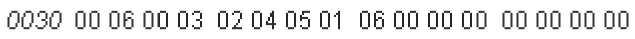
Does it run code? Figure. 87. Shows 6502 assembly code created by (6502 Software Desing, 2000). This program is a bubble sort it uses memory location 31 to store the length of a list, location 32 to hold a variable that indicates if any changes have been made, and location 33+ to store the list that should be sorted.



```
SORT8     LDY #$00      ;TURN EXCHANGE FLAG OFF (= 0)
          STY $32
          LDA ($30),Y   ;FETCH ELEMENT COUNT
          TAX           ; AND PUT IT INTO X
          INY           ;POINT TO FIRST ELEMENT IN LIST
          DEX           ;DECREMENT ELEMENT COUNT
NXTEL     LDA ($30),Y   ;FETCH ELEMENT
          INY
          CMP ($30),Y   ;IS IT LARGER THAN THE NEXT ELEMENT?
          BCC CHKEND
          BEQ CHKEND
                        ;YES. EXCHANGE ELEMENTS IN MEMORY
          PHA           ; BY SAVING LOW BYTE ON STACK.
          LDA ($30),Y   ; THEN GET HIGH BYTE AND
          DEY           ; STORE IT AT LOW ADDRESS
          STA ($30),Y
          PLA           ;PULL LOW BYTE FROM STACK
          INY           ; AND STORE IT AT HIGH ADDRESS
          STA ($30),Y
          LDA #$FF      ;TURN EXCHANGE FLAG ON (= -1)
          STA $32
CHKEND    DEX           ;END OF LIST?
          BNE NXTEL     ;NO. FETCH NEXT ELEMENT
          BIT $32       ;YES. EXCHANGE FLAG STILL OFF?
          BMI SORT8     ;NO. GO THROUGH LIST AGAIN
          RTS           ;YES. LIST IS NOW ORDERED
```

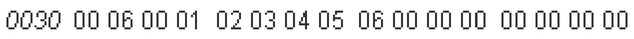
Fig. 87. 6502 Assembly Bubble Sort Program made by (6502 Software Desing, 2000).

Figure. 88. Shows memory location 0030-003F containing a size six list: 03, 02, 04, 05, 01, 06. After executing the bubble sort program this list should sort to contains values 01, 02, 03, 04, 05 and 06 in that order.



```
0030  00 06 00 03  02 04 05 01  06 00 00 00  00 00 00 00
```

Fig. 88. Test Input List.



```
0030  00 06 00 01  02 03 04 05  06 00 00 00  00 00 00 00
```

Fig. 89. Test Output List.

Clearly the FPGA can execute these instructions in this context. Completing one objective of this project “Successfully executing example 6502 code, comparing the outcome to the expected example, and finding no discernable differences”. However, we cannot conclude that this project has produced a circuit capable of emulating the 6502. Hardware verification at this scale is difficult, when designing the original 6502 it was found some instructions differed from their formal descriptions due to unforeseen fringe interactions usually with page boundaries in memory. Further research in hardware verification is required.

Is it Efficient? At the beginning I stated this project would be successful if when “Comparing the average execution time of instructions to their execution time when processed by and original 6502 with a decrease in time efficiency no greater than 20%”. In retrospect I was incredibly overconfident. The original 6502 produced by MOS technologies completes instructions in 2-5 ticks. A tick being one activation of any clock state. To calculate the time required for this project to execute an instruction you would multiply the number of clock cycles required by four, one for each clock state. This results in the smallest instructions completing in 4 ticks and the longest completing in 36 ticks and decrease in time efficiency by 100% to 720%. The results show that currently I cannot design processors as efficiently as Chuck Peddle and his team could in 1973. I believe there are three redundancies causing the largest reductions in time efficiency.

First. The longest Instruction BRK takes nine execution states to store three separate values into stack. However, one of these execution states stores the contents of the stack pointer in a stack buffer register which was described as loading a value into memory. This step is redundant as it would be simpler to read directly from the stack pointer. Implementing this change would result in all instruction that write to the stack taking 33.3…% fewer clock cycles to complete.

Second the clock has four clock states, with two states dedicated to decoding the initial idea was a 16-bit address could be retrieved in a single clock cycle. However, the test example in Figure. 87. uses no absolute addresses. This is a precedent amongst the majority of 6502 programs, with programmers preferring faster 8-bit addresses when possible. Therefore, the decode(highbyte) clock state is rarely used. Removing It would result in all instructions taking 25% less time to execute unless they require a 16-bit address.

Finally, the core design of this FPGA which relies on clock cycles is fundamentally flawed., There is never a situation during a 6502 instruction where an execution state triggers a decode state and likewise with decode and fetch. Due to this there is no need to cycle between all the states to see if anything must be done during each one. Instead, it would be more efficient to just trigger decode states until ready to execute. This would make short instructions like DEX take 2 ticks instead of a clock cycle, a 50% decrease in time. And long instructions like BRK to take 10 ticks instead of 9 clock cycles which is a 73.7…% decrease in time required.

Additionally, while not in the initial aims and objective the 6502 designed during this project is not wire efficient. For example, it is common practice to put all adders, subtractors and logical masks within an ALU. Instead of doing addition within a specific component the input numbers are sent to the ALU and then returned to the component. This is because these circuits are large and costly to manufacture. The current design of this FPGA contains 30 adders and subtractors with the arithmetic component holding the record for most adders and subtractors, containing thirteen.

12 Conclusion

In conclusion while this project has successfully accomplished many of its goals. Future opportunities include:

First implement custom RAM component currently this FPGA if printed onto silicon would not be able to function as a 6502. When designing RAM components their contents are sectioned this was seen in a lesser degree in this project with sections for the zeropage and the stack but typically there is a section designated for the operating system with a direct connection to ROM and a section connected to whatever I/O controller is attached called VRAM where user inputs are stored. For this FPGA to be able to execute complex code like the NES operating system a RAM component capable of housing them is required.

Second use this project to explore hardware verification techniques it is not possible to conclude this FPGA will perform as intended in every circumstance and while not that big problem for conventional software development printing chips that don't work has catastrophic outcomes for users and the company behind the design. A formal set of example programs, test vectors or logical proofs would alleviate this risk.

Finally a fundamental redesign of the clock as well as minor redesign to components like the Arithmetic Instruction Component would cause a significant improvement to the potential cost and efficiency of this FPGA if ever printed

13 References

6502 Software Design. Initial (2000) *Bubble Sort (for 8-bit Elements).* [online] Available at: <http://www.6502.org/source/sorting/bubble8.htm> [Accessed: 3 April 2024].

andkorzh, HardWareMan, org (2022) *Breaking NES Book.* [pdf] Available at: <https://github.com/emu-russia/breaks/releases/download/ppu-book revB8/Breaking_PPU_B8_Eng.pdf> [Accessed 22 October 2023].

Bagnall, B. (2006). *On the Edge: The Spectacular Rise & Fall of Commodore.*

Carr, J (1984*) 6502 User's Manual* [pdf] Available at: <https://ia902208.us.archive.org/23/items/6502um/6502UsersManual.pdf> [Accessed 30 October 2023].

Cox, R. (2011). research!rsc: *The MOS 6502 and the Best Layout Guy in the World*. [online] Available at: <https://research.swtch.com/6502> [Accessed 29 October 2023]

Leventhal, L.A. (1986). *6502 Assembly Language Programming*. McGraw-Hill/Osborne Media.

MOS Technology (1976) *KIM-1 MICROCOMPUTER MODULE USER MANUAL* [pdf] <Available at: https://dn790004.ca.archive.org/0/items/KIM-1_Users_Manual/KIM-1_Users_Manual_text.pdf> [Accessed 21 January 2024]

MOS Technology (1976) *The KIM 1 Microcomputer System* [pdf] Available at: <https://www.commodore.ca/wp-content/uploads/2013/02/brochure-kim1_.pdf> [Accessed 21 January 2024]

Self, G. (2019) *Logisim-Evolution Lab Manual*. [pdf] Available at: <https://cdn.hackaday.io/files/1814287762215552/logisim.pdf> [Accessed 29 October 2023]

Usborne Computer Books (1983) *Usborne Introduction to Machine Code for Beginners* [pdf] Available at: <https://archive.org/details/machine-code-for-beginners/mode/2up > [Accessed 1 February 2024]